2015

A Distributed Diffusion-Driven Algorithm for Load Balancing in an Electrical Power Grid

Isaac Abhilash Ayyala
Louisiana State University and Agricultural and Mechanical College, iayyal1@tigers.lsu.edu

A DISTRIBUTED DIFFUSION-DRIVEN ALGORITHM FOR LOAD BALANCING IN AN ELECTRICAL POWER GRID

A Thesis

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering

in

The Division of Electrical and Computer Engineering

by
Isaac Abhilash Ayyala
B.E., Visvesvaraya Technological University, 2012
May 2015

Acknowledgments

I would like to thank Dr. Ramachandran Vaidyanathan for giving me the opportunity to research in the field of distributed computing, and guiding me throughout my thesis in every step. Special thanks go to Dr. Shahab Mehraeen for his guidance and help with Power systems and other aspects of this thesis. Special thanks also go to Dr. Supratik Mukhopadhyay for his guidance and help with this thesis.

I thank my parents for supporting me morally and financially towards the completion of my Master's program.

I thank all my friends and colleagues who have helped me in this program. It was truly a good learning experience.

Table of Contents

Abstract

In this thesis we propose a distributed algorithm, based on diffusion, to balance loads on an electrical power grid, while maintaining stable operation (system's ability to maintain bus voltages within preset bounds). This algorithm, called the Diffusion-driven Distributed Load Balancing (DDLB) algorithm, is implemented on the OMNET++ Discrete Event Simulator and the response of the physical grid is simulated on a load flow program, which together simulate a deployment of the DDLB algorithm on the grid.

The electrical grid is represented as a graph whose nodes are buses and whose edges are power lines connecting buses. Each node (except the slack bus) has a load (positive if power consumed, negative if power generated) associated with it. The slack bus is a special bus that covers any power surplus or deficit due to a load assignment. A given load assignment, when applied to the grid affects bus voltages and system stability. The problem we address is as follows. Given a preferred load for each node and a load cost (a measure of deviation from this preferred load), the ideal solution is a load assignment with lowest cost that results in a stable system.

We measure the performance of our algorithm (DDLB) against the "one-shot" algorithm, a naive distributed solution in which each node uses its preferred load directly for a load assignment, without any regard for system stability. Through extensive simulations with 1.6 million test cases, we show that the DDLB algorithm vastly outperforms one-shot. Specifically, the one-shot algorithm causes instability in over 57% of the cases tested albeit with zero load cost. For the same cases when applied to the DDLB algorithm only 0.65% were unstable; the average load cost was less than 2%.

Our simulations included a study of several scenarios that a grid could be subjected to, including balanced load, overloaded, underloaded grids, local generator failures, and a sparser communication network for the DDLB algorithm; in this context one could view the one-shot algorithm as a distributed algorithm with no communication network. In all these scenarios studied the DDLB algorithm outperforms the one-shot algorithm.

1

Introduction

The power grid has evolved since its inception and is ever changing [15, 31]. In earlier times, the power grid was manually controlled from a central location [33], with human coordination responsible for maintaining a stable working system. With the advent of new power generation and storage technologies, even consumers at the "home" level, have started to play a somewhat different role in the overall power system in their capacity to act as generators. Having a central control is not enough to manage all the new entities spread out over the power grid. Households can own gas powered generators, solar panels [10], etc., that connect to the grid and could act as generators. Even more broadly, the power grid has diverse, and sometimes unpredictable, sources (such a wind farms, solar panels and, more futuristically, wave power) and load profiles (such as those with different patterns of use in an "intelligent home").

In order to make optimum use of these power sources and to obtain a solution that scales with the increased number of entities in the system, a distributed control (as opposed to centralized control) is required. This could be implemented using an underlying distributed computing network over which information about the electrical grid's load and stability could be exchanged and acted on. This control will make sure that as sources and loads are added and removed, the network will dynamically adjust and balance loads without any centralized control. A centralized control is not practical for such environments, since the communication and computational overhead would be massive if a central location was to be connected to every node in the network.

In this thesis we develop a distributed algorithm for load balancing on the electrical grid. A *distributed algorithm* is one without centralized control, in which individual computing nodes communicate with their neighbors, acting only on local information, to paint a global picture. We use a *synchronous* distributed algorithm, in which all computing nodes proceed lock step in rounds. Each round consists of a send-receive (communicate) and compute step. During the send-receive step, computing elements exchange information with their neighbors. In the compute step, a computation is performed based only on locally available information. In a synchronous environment, each node proceeds to the next communicate-compute step only after the current communicate-compute step has be completed at all nodes. Examples of distributed systems include the Internet, cloud based systems, sensor based networks and even networks on a chip (NoCs) [5, 20].

The distributed algorithm we use in this thesis is inspired by *diffusion algorithms* that are an example of synchronous distributed algorithms. In a diffusion algorithm, each node i starts with a local value v_i. The goal is to change this value to be the average $\frac{1}{n}\sum_i v_i$ of all values in the network, where n is the network size. This is done on a synchronous distributed system by slowly moving (diffusing) values from higher valued nodes through their neighbors to those with lower than average values in the network. More precisely if a node i and its neighbors have values v_i and v_j, then i sends a value proportional to $v_i - v_j$ to j. There are two factors that play a significant role in diffusion, convergence and stability. If the diffusion rate (the amount of the distributed "value" transacted in each round) is too slow then convergence will be very slow as well, whereas if the diffusion rate is too fast, then there could be stability issues, in which a node oscillates between high and low values.

Our algorithm is similar to diffusion because it also runs in rounds (or synchronous iterations), locally exchanges information and adjusts "values" in the search of a globally stable point. But there are major differences between diffusion and our algorithm. The values used in our case are not necessarily "zero-sum" quantities as in diffusion.

In diffusion, the reduction in the value at a node corresponds to the increase of the value among a subset of its neighbors; this is not necessarily the case for us. More importantly, in a diffusion algorithm a node acts independently and has no difficulty sending/receiving the value(s) prescribed by the algorithm. For example if i decides to send $\alpha = c(v_i - v_j)$ to j then is does so and the values v_i and v_j are adjusted to become $v_i - \alpha$ and $v_j + \alpha$, respectively, as intended by the algorithm. In our case, the value prescribed may not be controllable by a single node. For example, a node wanting to increase its value by a certain amount may not be able to do so unilaterally, as the underlying physical system behavior depends on all the nodes. That is, node i asking to increase its voltage by α may not result in the value actually increasing by α. Often the broad direction in which a part of the power grid may behave can be reasonably predicted. For example, an increase in voltage cost (cost used to express the voltage deviation from the nominal voltage) can generally be lowered by increasing the reactive power. However, the degree to which the system leans in the given direction is much more difficult to predict. Our algorithm is based on a simple use of these levers to adjust two cost metrics: the load cost and the voltage cost, that we explain later.

Our approach captures the interaction between two, somewhat separate, systems the physical power grid, and an overlaid distributed computing system. The (power) grid plane is a graph $\mathcal{G}_p(V, E)$, where V is a set of buses, and E is a set of power lines connecting the buses. The compute plane is another graph $\mathcal{G}_c(V, E')$, where V is the same set of nodes (but here they represent computational nodes rather than buses), and E' is a set of communication links between the computing nodes. Before we proceed, a few informal definitions are needed.

Let the network have n nodes numbered $1, 2, \ldots, n$. A *load vector* $\vec{\ell} = \langle \ell_i : 1 \leq i \leq n \rangle$ is a vector of (active) powers at each node i. That is, ℓ_i is the power flowing through node i. Each load vector $\vec{\ell}$ corresponds to a *voltage vector* $\vec{v} = \langle v : 1 \leq i \leq n \rangle$; when the load in ℓ is presented to the given power grid, then node i has a voltage v_i. Informally the pair $\langle \vec{\ell}, \vec{v} \rangle$, where $\vec{\ell}$ and $\vec{v}$ correspond to each other is called a *state of the system*. A

load profile $\lambda_i = \langle \lambda_{i,-1}, \lambda_{i,0}, \lambda_{i,+1} \rangle$, where $\lambda_{i,-1} \leq \lambda_{i,0} \leq \lambda_{i,+1}$ for a node i defines the acceptable load range of the node. The load $\lambda_{i,0}$ is the *preferred load* of node i. However the node shows *load flexibility* in that it finds any load ℓ_i such that $\lambda_{i,-1} \leq \ell_i \leq \lambda_{i,+1}$ acceptable. The *voltage profile* $\mathcal{V}_i = \langle \mathcal{V}_{i,-1}, \mathcal{V}_{i,0}, \mathcal{V}_{i,+1} \rangle$ of node i and its *voltage flexibility* are defined similarly.

A system state $\langle \vec{\ell}, \vec{v} \rangle$ is *stable* (with respect to a given voltage profile) $\mathcal{V}$ iff for all $1 \leq i \leq n$, $\mathcal{V}_{i,-1} \leq v_i \leq \mathcal{V}_{i,+1}$, where $\vec{v} = \langle v_i : 1 \leq i \leq n \rangle$. The cost of state $\langle \vec{\ell}, \vec{v} \rangle$ with respect to a load profile $\vec{\lambda}$ is denoted by $c(\vec{\ell}, \vec{v})$. This quantity expresses a measure of the deviation of ℓ_i from $\lambda_{i,0}$, particularly outside the range $[\lambda_{i,-1}, \lambda_{i,+1}]$; here $\vec{\ell} = \langle \vec{\ell}_i : 1 \leq i \leq n \rangle$.

Given a stable state $\langle \vec{\ell^0}, \vec{v^0} \rangle$ and load and voltage profiles $\vec{\lambda}, \vec{\mathcal{V}}$, the problem is to find a stable state $\langle \vec{\ell}, \vec{v_\ell} \rangle$ such that $c(\vec{\ell}, \vec{\lambda})$ is small. This is a difficult (often impossible) problem, particularly when $\vec{\lambda}$ represents an "unbalanced system."

For a given profile $\vec{\lambda} = \langle \lambda_i : 1 \leq i \leq n \rangle$, where $\lambda_i = \langle \lambda_{i,-1}, \lambda_{i,0}, \lambda_{i,+1} \rangle$, let $\lambda_0 = \langle \lambda_{i,0} : 1 \leq i \leq n \rangle$ be the preferred load vector. One solution to the problem described above to start at state $\langle \vec{\ell^0}, \vec{v^0} \rangle$ and simply ask the system to go to state $\langle \vec{\lambda}_0, \vec{v_{\lambda_0}} \rangle$. We call this the *one-shot method*. While this results in $c(\vec{\lambda}_0, \vec{\lambda}) = 0$ cost, it often results in an unstable state. In contrast, our distributed solution, which accounts for system stability, performs much better as discussed in the next section.

The algorithm we propose is called the Diffusion-driven Distributed Load Balancing (DDLB) algorithm.

We use the OMNET++ discrete event simulator to model the compute plane, and a load flow analysis program to simulate the grid plane. Inputs are generated for different scenarios, such as "balanced", "overloaded", "underloaded", "subgraph", and "radial damage"; details appear later. Each of these scenarios is also tested with zero load-flexibility (where $\lambda_{i,-1} = \lambda_{i,0} = \lambda_{i,+1} = 0$) and zero voltage-flexibility (here $\mathcal{V}_{i,-1} = \mathcal{V}_{i,0} = \mathcal{V}_{i,+1}$). Each input in a scenario represents a new state that the power grid would like to move to from a (current) stable state. If the new state is directly ap-

plied to the power grid (one-shot algorithm), stability may not be achieved. We apply the DDLB algorithm, which applies new loads in small increments in iterations to the power grid, trying to keep it stable at each step. In each iteration, the new loads are calculated based on costs that are computed from the information exchanged between neighboring nodes.

1.1 Key Findings

We have previously argued the benefits of a distributed solution over the centralized approach in solving load balancing on the grid. The main contribution of this thesis is to demonstrate that the proposed DDLB algorithm has significant advantages over the one-shot algorithm. Specifically we show the following based on around 1.6 million simulated cases.

1. On a average (over a range of grid scenarios) the DDLB algorithm finds a stable solution in over 99% of the cases simulated (at the expense of an average increase of less than 2% in the load cost). For the same input instances, the one-shot algorithm finds a stable solution in only about 43% of the cases.
2. Even in overloaded situations (where the power supplied by the grid falls short of the load demand), the DDLB algorithm fails to attain a stable state in less than 2% of the cases (as opposed to over 58% for the one shot algorithm). Moreover for these cases, the load error of the DDLB algorithm can be held to around 15%, a small price to pay for the large gains in stability.
3. In an underloaded case (where the supply exceeds the demand), the DDLB algorithm results in an unstable system in less than 0.3% of the time (in contrast to over 58% for one-shot). Moreover, the load error is around 0.4%. For this case, the DDLB algorithm seems to be superior to the one-shot algorithm in almost all respects.
4. For the balanced case, when power supply and loads are about the same, the DDLB algorithm has 0.12% of the cases being unstable. Surprisingly, for this case,

the one-shot algorithm reaches a stable state for less than 55% of the tested inputs. The load error for the DDLB algorithm is almost 0.

5. One advantage of the DDLB algorithm over the one-shot algorithm is that the former includes internode communication as it navigates the solution space, whereas the latter makes an (unguided) beeline for the required loads. We removed about a third of the edges in the communication topology used for the DDLB algorithm (based on edge length or internode distance). The results were not significantly different from the full topology. The DDLB algorithm failed (reaching an unstable state) for only about 0.56% of the the total cases while the one-shot algorithm failed in about 48% of the cases. Thus, the DDLB algorithm appears to be quite robust with respect to its dependence on the communication network.
6. We studied cases where the topology suffers localized damage to its sources (which are incapable of supplying power after the damage). Here too the DDLB algorithm outperforms the one-shot algorithm, reaching stability in over 99% of the cases while the one shot algorithm reaches stability in only about 45% of the cases. The load error of the DDLB algorithm is less than 2%.
7. In general, the one shot algorithm is better than the DDLB algorithm for the load cost, which is 0 for the one-shot algorithm. However, this comes at the price of substantially lower stability. The slack bus usage (to mitigate power imbalances) is somewhat mixed. The one-shot algorithm is slightly better than the DDLB algorithm (with respect to the slack bus use) for all but the balanced case, where the DDLB algorithm is slightly better.

On the whole the DDLB algorithm shows much promise.

1.2 Literature Review

Many centralized, distributed and hybrid control schemes have been proposed to reduce the impact of load distribution on the transmission system voltages and to improve stability.

The centralized approach of Kazari et al. [18] uses genetic algorithms to control the reactive power output, substation capacitors and tap settings of transmission transformers. The centralized coordination needs an infrastructure including remote sensors and actuators, along with communication links to the central node. Kiani and Annaswamy [19] propose a semi-centralized approach using a hierarchical model where the dynamics of the grid are controlled at the primary, secondary, and tertiary levels. Other approaches perform intentional islanding to increase the system reliability [32] or use one central control per microgrid [29]. Other centralized approaches exist [6, 27]. All these approaches involve heavy communication overheads as the information has to be sent to a central location one way or the other. Such centralized approaches therefore do not scale to very large systems.

Some of the distributed load management methods that have been proposed include autonomous demand side management through the deployment of energy consumption scheduling (ECS) devices in smart meters within a neighborhood [22]. Others, based on the capacity of users to manage their own demand in order to minimize a cost function that is based on price of other users actions [16]. User load profiles are used for dynamic pricing schemes that are used to incentivize consumers to contribute to an aggregate load profile suitable for utilities [8]. Distributed algorithms requiring only some limited message exchanges between users have been proposed where each user tries to maximize its own benefits in a game-theoretic setting [23]. Other approaches also appear in the literature [7, 24, 21, 25].

Some studies propose a hybrid demand-response policy (straddling both the centralized and distributed policies) in which most of the decisions are taken on local smart meter devices, and some of the requests are passed on to the central location unit for centralized control [2]. However these are, although to a smaller extent, also subject to scaling difficulties. Another direction is a scheduling and load balancing algorithm for an N-resource node grid that not only takes into account the heterogeneity in the nodes and network , but also considers the overhead involved in coordinating

among the nodes [3]. In other approaches, loads are classified into soft loads (shiftable in time) and hard loads (non shiftable), and the power consumption of a community is adjusted by shifting the soft loads to off-peak hours in a probabilistic way that results in a relatively constant overall power consumption profile [28]. This is somewhat similar to the flexible loads we consider in this thesis, in which a node requests a load $\lambda_{i,0}$ but is willing to accept a value between a lowerbound $\lambda_{i,-1}$ and upperbound $\lambda_{i,+1}$, where $\lambda_{i,-1} \leq \lambda_{i,0} \leq \lambda_{i,+1}$.

Conventional control methodologies do not focus on the communication between nodes, rather they involve robust predictive current control to enhance the system robustness and reduce the current distortion due to the control delay and inductance deviation [12]. Automatic generation control (AGC) is the most widely used control strategy and its primary purpose is to balance the total system generation against system load and losses so that the desired frequency and power interchange with neighboring systems are maintained [26].

The DDLB algorithm discussed in this thesis is inspired by diffusion algorithms [1, 4, 11, 17]. Our algorithm has a diffusion type of structure in which each node compares cost metrics (voltage cost, load cost) with its neighbors and takes action to diffuse the cost within the neighborhood. More details on our approach appear in Chapters 3 and 4.

1.3 Organization of this thesis

Chapter 2 describes some power system terminology, and introduces the OMNET++ and load flow analysis programs we used to carry out the simulations. Chapter 3 gives a brief overview of the structure of our algorithm and its interaction with the simulated power system. Chapter 4 discusses the DDLB algorithm, which is the focus of this thesis, and its implementation. Chapter 5 details the simulation framework including the generation of inputs for the various scenarios studied. Chapter 6 presents the results obtained for the different scenarios. Chapter 7 gives the conclusions from this thesis and recommendations for further work.

2

Preliminaries

In this chapter we discuss some preliminary ideas about power systems and outline two software packages employed in our work (a load flow analysis program and the OMNET++ Discrete Event Simulator [30]).

2.1 Power System Terminology

The terminology described here is used in the load flow program (explained in Section 2.2) and in the proposed distributed algorithm detailed in Chapter 4.

A power grid is a system of interconnected buses (loads and generators). The *grid topology* is a graph that models these interconnections between buses. Consider the example grid topology shown in Figure 2.1. This topology will be used as a running example to explain all the terminology introduced in this chapter. The network of Figure 2.1 has 8 nodes. A *node* (or *bus*) is either a load or a generator, capable of acting as a power sink or source, respectively; the terms "node" and "bus" are used interchangeably in the context of the "(power) grid graph" throughout this thesis. A *line* represents a power line connecting two or more buses. There are three kinds of buses in a typical power system - load bus, generator bus and slack bus. A *load* bus, as the name suggests, is a power sink; that is, it consumes power. A *generator* bus, on the other hand, feeds power into the system. Another way of defining load and generator buses is as follows. A bus without any generators connected to it is called a load bus, a bus with at least one generator connected to it is called a generator bus. The *slack* bus is a special bus that can act as both a load or generator as the situation demands. It absorbs or provides for the excess or deficiency of power in the network. It either generates excess power if needed or consumes excess power by acting as a load. In the example

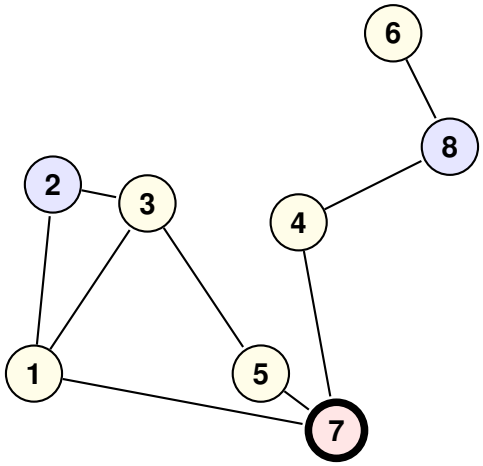


FIGURE 2.1: An example of an electrical grid topology. The shaded buses are generators and bus 7 in bold is the slack bus, other buses are loads

of Figure 2.1, we will let nodes 2 and 8 represent generator buses. Let node 7 be the slack bus and let the remaining buses be load buses. The term load (or power) is used to indicate the power consumed at a node. A negative load is the power generated at the node.

TABLE 2.1: Example of bus information

node i	V_i	bus type	P_i	Q_i
1	1.000	3	0.012	0.002
2	1.033	2	-0.400	-0.080
3	1.000	3	0.124	0.024
4	1.000	3	0.076	0.003
5	1.000	3	0.2	0.040
6	1.000	3	0.2	0.040
7	1.040	1	0	0
8	1.010	2	-0.400	-0.080

For a given grid topology the *bus information* constitutes for each node i, its nominal voltage V_i (this is the same as $\mathcal{V}_{i,0}$), bus type (slack (1), generator (2) or load (3)), and load (active power P_i, reactive power Q_i). The power is negative for generators and positive for loads. Table 2.1 shows an example of bus information. The power system uses additional parameters, including line resistance (R), reactance (X) and susceptance (B). These together form the *line information* (shown in Table 2.2 for the nodes in the example of Figure 2.1). The load flow program uses additional quantities in its bus and line information. These include voltage angle in the bus information, and tap ratio

(a transformer parameter) and phase angle in the line information (the tap changes the transformation voltage ratio. It can be used to slightly modify the voltage by increasing it when voltage is low. We did not use the tap ratio in this work, as the aim is to affect load balance through local changes). The bus information and line information form the inputs to the load flow analysis program that is discussed in Section 2.2.

Comparing Figure 2.1 and Table 2.1 we can see that node 7 (the slack bus) is assigned 0 for its loads. Due to the potential for voltage drops, generator and slack bus voltages are generally kept slightly higher than the loads. Also in Table 2.2 it is seen that the first column, edge (a, b) (see also Figure 2.1), represents the edge in the grid graph. The line information elements $R(a, b)$ (resistance), $X(a, b)$ (reactance) and $B(a, b)$ (susceptance) are also shown.

TABLE 2.2: Example of line information

edge (a, b)	$R(a, b)$	$X(a, b)$	$B(a, b)$
(1 , 2)	0.0192	0.0575	0.0264
(1 , 3)	0.0452	0.1852	0.0204
(2 , 3)	0.0132	0.0379	0.0042
(1 , 7)	0.0472	0.1983	0.0209
(3 , 5)	0.0581	0.1763	0.0187
(5 , 7)	0.0119	0.0414	0.0045
(4 , 8)	0.0460	0.1160	0.0102
(4 , 7)	0.0267	0.0820	0.0085
(6 , 8)	0.0120	0.0420	0.0045

A *per-unit* (pu) system is one in which each system parameter is expressed normalized with respect to a standardized/normal value for that parameter. For example, the normal voltage of a bus in Table 2.1 is assumed to be 1. The nominal voltage could be 110 V. If the actual voltage is 100V then the normalized voltage would be $100/110 = 0.909$. A system tolerance of $\pm 5\%$ allows the normalized voltage to vary between 0.95 and 1.05. In the same way, load is also expressed normalized. Here 1 pu of load equals 100 MW.

2.2 The Load Flow Analysis Program

A key program used to simulate a physical power grid is the load flow analysis program. It is implemented in C and is used to determine node voltages of the power system, given its topology, bus information and line information (discussed in Section 2.1). It also produces other quantities such as the line flow matrix, but we do not use these quantities.

The program is based on the power flow solution, which is an approach to solve the power flow equations for an interconnected system (see Equations (2.2.1) and (2.2.2)). The number of nodes in the network is n. For a node i, P_i and Q_i represent its active power and reactive power, respectively. The line admittance (of the edge) between node i and node k is $Y_{i,k}$. The node voltage of node i is V_i and its voltage phase angle is δ_i. Then

$$P_i = \sum_{k=1}^{n} |Y_{i,k} V_i V_k| \cos(\theta_{i,k} + \delta_k - \delta_i) \tag{2.2.1}$$

$$Q_i = -\sum_{k=1}^{n} |Y_{i,k} V_i V_k| \sin(\theta_{i,k} + \delta_k - \delta_i) \tag{2.2.2}$$

Here angle $\theta_{i,k}$ is the admittance angle between nodes i and k, derived from Equation (2.2.3) below for the line admittance.

$$Y_{i,k} = |Y_{i,k}| \angle \theta_{i,k} = |Y_{i,k}| \cos \theta_{i,k} + j |Y_{i,k}| \sin \theta_{i,k} = G_{i,k} + jB_{i,k} \tag{2.2.3}$$

where $j = \sqrt{-1}$

Equations (2.2.1) and (2.2.2) use node i and its neighbor k. However Equation (2.2.3) holds for any pair of nodes i and k (that are not necessarily neighbors on the grid).

The principal information obtained from the power-flow solution is the magnitude and phase angle of the voltage at each bus (node), and the active and reactive power flow in each line (edge). Figure 2.2 shows the block diagram of the load flow program, which takes the desired active power $\vec{P_0} = \langle P_{0,i} : 1 \leq i \leq n \rangle$ and reactive power $\vec{Q_0} = \langle Q_{0,i} : 1 \leq i \leq n \rangle$ and the grid topology $\mathcal{G}_p$, and produces the node voltages $\vec{V} = \langle V_i : 1 \leq i \leq n \rangle$ and achieved loads $\vec{P} = \langle P_i : 1 \leq i \leq n \rangle$ and $\vec{Q} = \langle Q_i : 1 \leq i \leq$

$n\rangle$. The desired loads $(\vec{P}_0, \vec{Q}_0)$ differ from the achieved loads $(\vec{P}, \vec{Q})$ only for the slack bus. Note also that the entire input can be represented as the line and bus information.

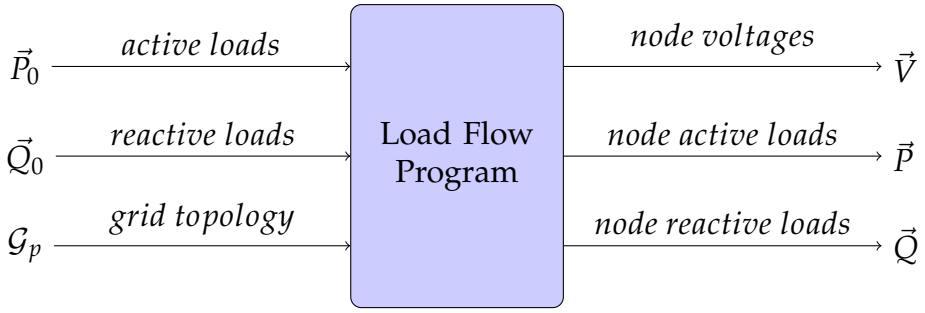


FIGURE 2.2: A schematic view of the load flow program

There are several different methods of solving the nonlinear system of equations that characterize the system (Equations (2.2.1) and (2.2.2)). These include the Gauss-Seidel, Newton-Raphson, fast-decoupled loadflow, holomorphic embedding load flow methods. The Newton Raphson method is the most popular among these and is the one used for the load flow program used in this thesis.

The load flow program runs in iterations and the stability of the system is determined based on its convergence to a solution. If there is no convergence for the state of the system presented as input, then the load flow program returns a normalized voltage of 0 or 2 (indicating the direction of the divergence). Moreover, the active and reactive loads are indicated as "nan" or "not a number".

The load flow program is used as a replacement for the physical system in our study; that is, as a simulation of the power grid, returning its response to various load changes at the nodes. The load flow program plays an important role in the distributed load balancing simulation. For the given desired active and reactive loads, the program returns the voltage feedback. This feedback is based on the loads and interconnections of various nodes. A slight change in topology or the loads can give a completely different voltage output when run through the load flow solution. This feedback information is used by the proposed DDLB algorithm to make corrections and changes to its load prescription for the next iteration. The DDLB algorithm works in iterations too, where the load flow program is called after each iteration to test the

stability of the new state of the power system (Chapter 3 discusses the relationship between our algorithm and the load flow analysis program).

A load flow study provides insight and recommendations for the system operation and optimization of control settings that helps to obtain maximum capacity while minimizing the operating costs. Voltage is the determining factor in the stability of a system, and it needs to be within a narrow range (for example $\pm 5\%$ of the nominal value). The feedback obtained from the load flow program helps the DDLB algorithm take the necessary action.

The DDLB algorithm uses the reaction of the physical system to the desired loads in an iteration to determine the action to be taken in the next iteration. In an actual deployment the true local values of parameters such as voltages and loads would be available to a node (Chapter 4). In our study the load flow analysis program is a stand-in for the physical network. While it is fairly accurate, it is not a full representation of the physical system.

There is a more powerful simulator of the power grid called ETAP [14]. It is a commercially available software, used for simulation, design and optimization etc. of the power grid. It could be used in future work to further test the effectiveness of our algorithm.

2.3 The OMNET++ Discrete Event Simulator

As noted earlier, the electrical grid is represented by a graph $\mathcal{G}_p$ of buses and links. This graph is called the *grid plane*. Also present is a graph $\mathcal{G}_c$ (whose nodes are computing elements) called the *compute plane*. While the grid plane is simulated by the load flow program, the distributed computational system (compute plane) is simulated by the OMNET++ discrete event simulator. The distributed load balancing simulation is run on this framework. The external load flow solution code (written in C) is integrated to work with the OMNET++ code so that at each iteration of the OMNET++ code (load balancing algorithm), the load flow program is called.

OMNET++ is a modular C++ simulation library and framework, primarily for building network simulators. It provides libraries to define networks (in our case the compute plane) and to write the functionality for each computing node in the network. It handles the message passing between the nodes; in our case, nodes operate synchronously in time. This results in a computation at the nodes being done in parallel (in the simulated time), although the code is run sequentially (in real time) on the simulating machine.

The OMNET++ framework requires the network to be first defined, which, in our case, is obtained from the line information (see Table 2.2). In Table 2.2 the edge (a, b) column is used to define the network. The distributed algorithm consists of a (common) code for each node; this is written in C++. The OMNET++ library provides modules for initialization of the network and message passing between the nodes. The main part of our algorithm involves message passing between computing elements containing information about the power grid nodes themselves (voltage, load, cost metrics); these are discussed in Chapters 3 and 4.

OMNET++ provides a message handling module called `activity(i)`, where i is the index of the node. Calling `activity(i)` in the OMNET++ framework causes every enabled node i to perform (simultaneously in simulated time) the activity (program segment) given in the module. This is explained in Figure 2.3.

Recall that we described a synchronous system as one in which each round involves a $\underbrace{\text{write-read}}_{\text{communicate}}$-compute sequence. Notice that by performing a synchronizing step after each write, a round may be viewed as a "read-compute-write" step.

The `activity(i)` module for each node i runs asynchronously in simulated time, performing a read-compute-write sequence in each round. Modules synchronize after they send their information and receive the results from the load flow program. Each compute node corresponds to an activity block (see Figure 2.3). The nodes execute the block independently and then synchronize at the end of each block. Consequently, in a round r they first receive (what was sent in the previous round $r - 1$), compute based

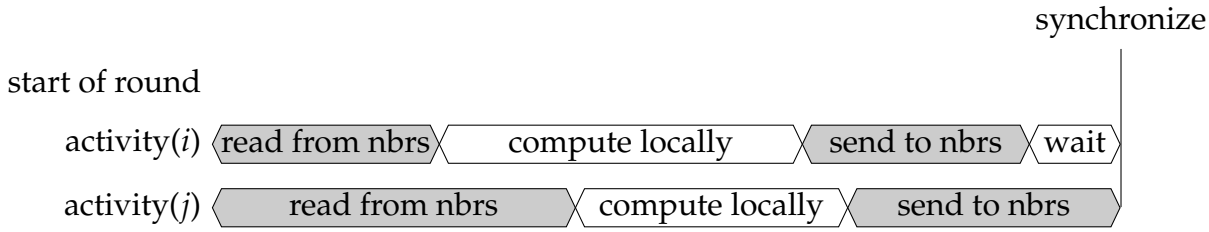


FIGURE 2.3: The timing of the `activity` module during one round

on local state and then send out a new set of information to their neighbors before the next receive (of round $r + 1$) is executed, nodes are synchronized so that all sends of round r have been completed.

The OMNET++ program is coded such that it requires only two input files to operate (bus information and line information, similar to Tables 2.1 and 2.2). The details of the input file creation and their structure is discussed in Chapter 5. For different scenarios, these input files are changed accordingly and fed into the simulation. The details of the algorithm behind this OMNET++ program is described in Chapter 4.

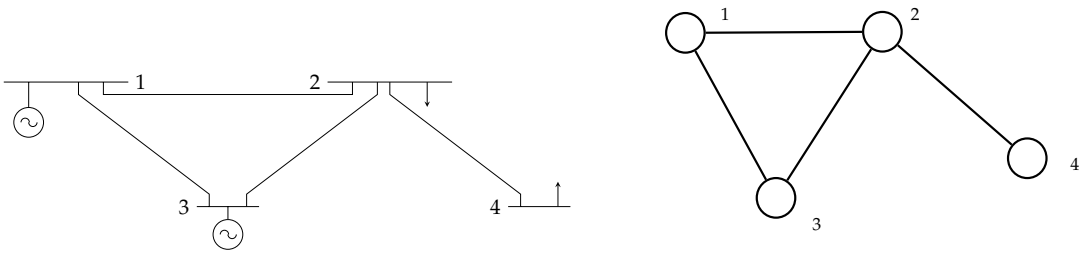
3

System Overview and Structure

In this chapter we describe how the proposed algorithm (DDLB), is implemented on the OMNET++ Discrete Event Simulator, and detail how it works with the load flow program algorithm to emulate the deployment of the DDLB algorithm on the compute plane of the physical grid. We also describe different grid scenarios studied in this thesis.

The usual representation of the power grid is as shown in Figure 3.1 (a) for a small example of four nodes. Here buses 1 and 3 are generator buses (indicated by the circles) and buses 2 and 4 are load buses. Figure 3.1 (b) shows a graph representation of this grid.

The physical infrastructure has the grid of buses and power lines, overlaid with the computing hardware and communication links between them. Different load demands (positive for load and negative for supply or generator) can be physically made on the power grid; the low level equipment needed for regulating the load and power supply are not shown in the figure. A corresponding compute plane has the same topology as the grid plane for most of this thesis; here nodes represent computing elements and edges represent communication links. In order to understand the feedback



(a) (b)

FIGURE 3.1: An example physical power grid (a) and its graph representation (b)

mechanism and the interaction between the simulation and the load flow program in our approach, we need to further discuss the ideas of the *compute plane* and the *grid plane* (that were briefly introduced in Section 2.3). The grid plane represents the actual physical network of connected buses, whereas the compute plane represents the communication topology connecting computational nodes. It is not necessary for these two planes to be identical, but let us assume that they are for this discussion. The two planes can be visualized as shown in Figure 3.2.

Each grid plane node (bus) has a computational node associated with it. This node calculates the local load and conveys it to the physical load regulating hardware. Each computational node also has a program running on it and it exchanges information with the neighboring computational nodes through the compute plane. It also receives information in the form of the response of the physical system; in our case this is the voltage. This is represented as the lower circle in the middle of Figure 3.2. The compute nodes collectively form a distributed computing system whose external inputs and outputs are from the physical power grid.

The same program (discussed in Chapter 4) is run on all compute nodes. After a round of computation is complete, each node provides a recommendation for grid action, and receives a feedback (containing the voltages, active and reactive powers) from the grid. This relationship is illustrated in the right "simulation" side of Figure 3.2.

This work assumes the grid to be small enough to respond quickly. More precisely, we assume the grid response to be comparable to a round of the computational plane. We elaborate on this further later in this chapter. Alternate approaches are suggested in Section 7.1. Our algorithm (in the compute plane) proceeds in rounds, and there is synchronization with the load flow analysis program (grid plane) after each round (see Section 2.3 and Figure 2.3). For each node i, a round includes reading the state of the neighborhood (that is, receiving the data from its neighbors), computing the costs based on local load and voltage and data received from neighbors, and giving a prescription to the grid plane node on what load to assume next. The load regulators

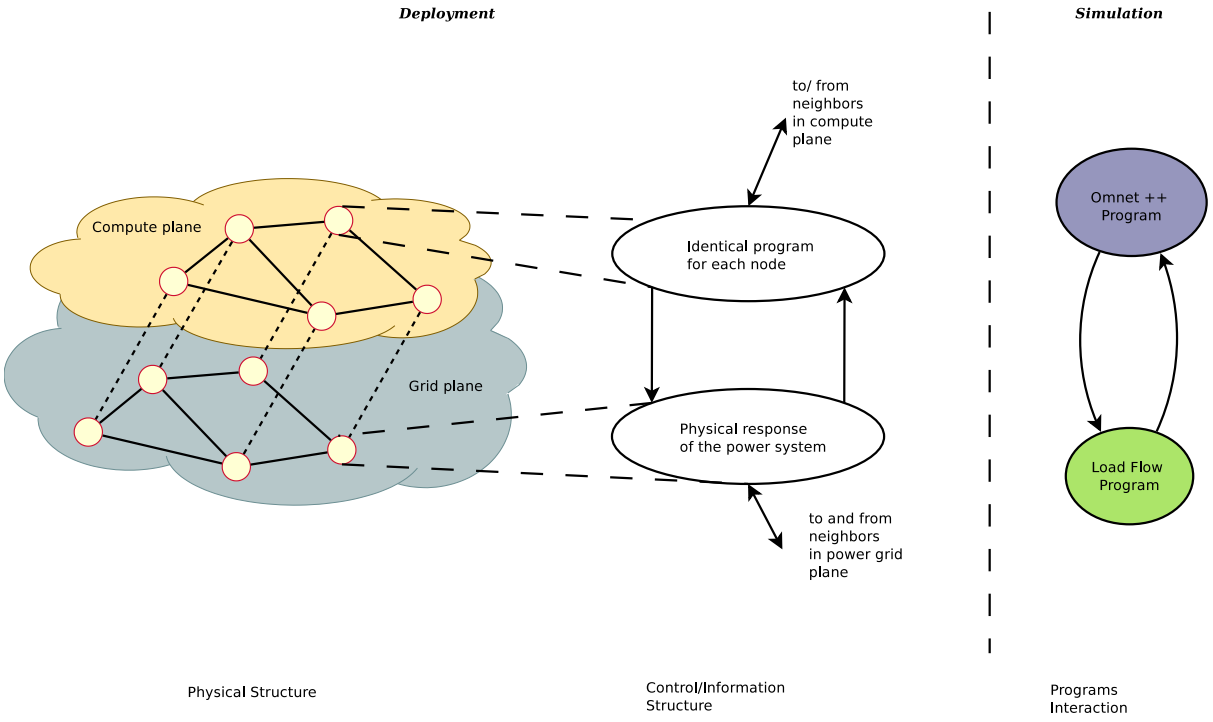


FIGURE 3.2: Relationship between deployment of a distributed load control and its simulation

in the grid plane handle the actuation of this prescription. After the prescription is actuated, the physical power grid responds, by taking into account all the node load actions across the network. This response may or may not match the compute node's expectation of voltage. This is because it is making a local decision (based on its neighborhood and its own value. However, the power flow in the physical grid results in a global effect in the physical grid. Each node i changes its load slowly in each round, using the grid response to compute the next prescription. This is different from the *one shot* algorithm, whose progress can be described as observing from a distance and moving towards the destination with "eyes closed", not fully understanding that the system causes a gap between theory and practice; this could lead to an off-target physical state where voltage stability is not satisfied. Using the same analogy, our approach is like moving with open eyes and after each small step making minor changes to the load on the way to the destination. The challenge with distributed load balancing in the power system is that we cannot see the global state from the local node and yet we must manage to achieve a desired global stable state.

The system aims to get a target state $S^f = \langle S_i^f : 1 \leq i \leq n \rangle$, where $S_i^f = \langle \mathcal{V}_i^f, \lambda_i^f, Q_i^f \rangle$; here $\mathcal{V}_i^f, \lambda_i^f, Q_i^f$ are the target voltage, active power and reactive power at node i. The voltage must satisfy the constraint $\mathcal{V}_{i,-1} \leq \mathcal{V}_i^f \leq \mathcal{V}_{i,+1}$. For our study the target load voltages are 1 (that is, $\mathcal{V}_i^f = 1$ for loads) and they allow a 5% flexibility. Therefore $\mathcal{V}_{i,-1} = 0.95$ and $\mathcal{V}_{i,+1} = 1.05$. For sources, there is no flexibility in voltage. Here $\mathcal{V}_{i,-1} = \mathcal{V}_i^f = \mathcal{V}_{i,+1} > 1$. Similarly, $\lambda_{i,-1}^f \leq \lambda_i^f \leq \lambda_{i,+1}^f$ represents the active power flexibility. The reactive power is typically unconstrained.

Let the power grid have n nodes $\pi_1, \pi_2, \ldots \pi_n$. Let node π_i correspond to compute node γ_i (where $1 \leq i \leq n$). Let the state of π_i at the start of round t (or end of round $t-1$) be $S_{i,t} = \langle V_{i,t}, P_{i,t}, Q_{i,t} \rangle$ where $V_{i,t}, P_{i,t}, Q_{i,t}$ represents for node i and round t, the voltage, active power and reactive power, respectively, flowing through node π_i. The state of γ_i at the start of round t is $\widehat{S}_{i,t} = \langle \widehat{V}_{i,t}, \widehat{P}_{i,t}, \widehat{Q}_{i,t} \rangle$, where $\widehat{V}_{i,t}, \widehat{P}_{i,t}$ and $\widehat{Q}_{i,t}$ represents for node i and round t, the voltage, active power and reactive power, respectively, that is recommended by the compute node γ_i. State $\widehat{S}_{i,t}$ represents the recommendation provided by the compute node to the physical grid and $S_{i,t}$ is the grid's response to this recommendation. Ideally $S_{i,t} = \widehat{S}_{i,t}$.

The compute node γ_i first obtains $S_{i,t}$ from its π_i. Then keeping S_i^f in mind it exchanges information about S_i^f and $S_{i,t}$ with its neighbors γ_j (that provide information about $S_j^f, S_{j,t}$). Using this information (collectively called $\widetilde{S}_{j,t+1}$) node γ_i computes a new prescription $\widehat{S}_{i,t+1}$ that it sends to π_i and awaits its response $S_{i,t+1}$ in the next round. This is illustrated in Figure 3.3.

Whenever the load of a node is changed in the power grid, there will be changes in voltages across the network, affecting not only that particular node but also its neighbors and other nodes in the network, depending on several factors like location and degree of the node. The new voltages of the grid due to the change in load take some time to settle down and this delay in feedback affects how we model the distributed algorithm. In our study, we assume this delay to be very small. Since our algorithm makes small changes at a time, a quick response is not unreasonable. Our algorithm

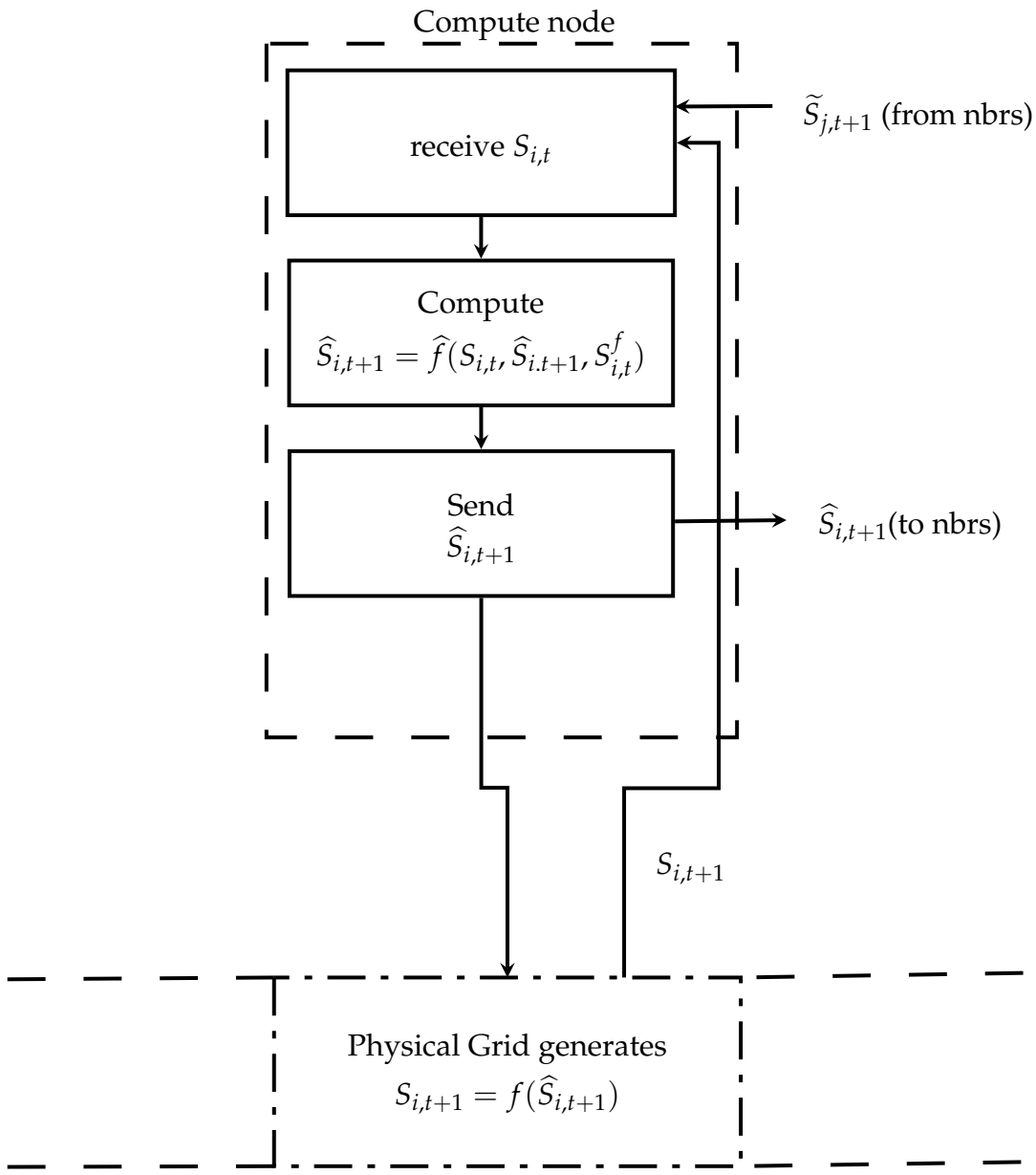


FIGURE 3.3: Compute plane and power plane interaction for a node

runs in iterations in which a feedback is needed from the physical system after each iteration. This is illustrated in Figure 3.4. The feedback is used to calculate the new load prescription for each node in a distributed manner.

The details of the costs and the calculation of the new prescription is described in Chapter 4. However we briefly describe the meaning of costs here. As mentioned earlier, along with the expected loads and voltages, the load and voltage flexibilities are given as input to the simulation. The load flexibility defines the region around the desired load value within which the load is acceptable. For example, if the asked load for

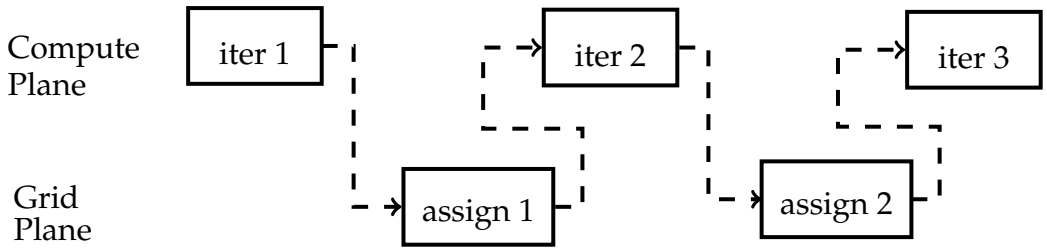


FIGURE 3.4: Time constraint assumption

a particular node is 2.5, and the load flexibility is 2.3 to 2.7, all the values between 2.3 and 2.7 are acceptable although it is preferred to have 2.5 as the load. Therefore, the load cost for this node is 0 at 2.5 and small if the load is within the range $[2.3, 2.7]$. It increases much more rapidly outside this range. Similarly, using the voltage flexibility, voltage cost is calculated.

3.1 Grid State Scenarios

To capture various situations the power grid may be subjected to, we study several scenarios that we now introduce. More details appear in Section 5.3. Recall that positive and negative powers represent loads and sources respectively, and the slack bus is a special bus that can act as both a load or generator as the situation demands. For the following let bus n be the slack bus.

1. Balanced : Here the loads and sources balance each other, and there is little load to be expected on the slack bus. $\sum_{i=1}^{n-1} P_i = 0$, where, P_i is the active power of node i. Typically losses in the power grid will cause it to be little different from 0.
2. Overloaded : Here there is excess load in the system, and the slack bus takes the burden of generating the excess power; here $\sum_{i=1}^{n-1} p_i > 0$.
3. Underloaded : Here the grid has excess generation of power and the slack absorbs the excess load; here $\sum_{i=1}^{n-1} p_i < 0$.
4. Subgraph : Here the compute plane topology is a subset of the grid topology. The subset is created based on "distances" between nodes.
5. Radial failure : This models failure of generators within radius (graph distance of 1 and 2) of a randomly selected node. It represents a local mishap.

These scenarios are simulated in our study of the DDLB algorithm.

4

The Diffusion-Driven Load Balancing Algorithm

In this chapter the details of the distributed diffusion-driven load balancing (DDLB) algorithm are discussed. The algorithm is, as the name indicates, inspired by diffusion algorithms. The input to a diffusion algorithm is a weight w_i for each node i, which is the same in the case of our algorithm as well, where the weights correspond to the active and reactive powers ascribed to each node. Our setting also includes the node voltage which is used as a measure of system stability. As explained below the voltage also plays a role in our setting that is similar to the weight in diffusion algorithms. In diffusion, nodes communicate to exchange weights so that the final weight is equal (well balanced) across the nodes. In our case the information about the load and voltages (expressed as load and voltage costs) are exchanged instead of the weight. The load and voltage costs are similar to the weight deviation $\frac{(|w_i - w_{average}|)}{w_i}$ that measures how far a node is from its stable state in diffusion. In our algorithm we are trying to minimize the loads and voltage costs for all the nodes, instead of the weight deviation. A diffusion algorithm works synchronously in rounds. In each round a node sends some of its weight to (or receives some weight from) each of its neighbors. Our algorithm works synchronously too, as explained in Chapter 3. Each compute node synchronously sends its new load prescription to the load flow program, which in turn, produces system voltages corresponding to the loads. This is used to check for system stability.

In diffusion, neighbors transact weights based on the difference of their weights, that is, if nodes i and j currently have weights w_i and w_j then the amount of weight transacted between node i and j is proportional to $w_i - w_j$. The direction of this trans-

action depends on the sign of $w_i - w_j$. In our algorithm, the local and neighborhood loads and voltages are distilled into two local quantities, the load cost LC_i and the voltage cost VC_i and the doublet (LC_i, VC_i) is used to guide "diffusion." More specifically if $Nbr(i)$ is the set of neighbors of i, then node i computes the quantity,

$$X_i = g\left(\langle LC_i, VC_i\rangle \cup \{\langle LC - j, VC - j\rangle \ : \ j \in Nbr(i)\}\right),$$

where g is a function of the neighborhood costs.

Unlike diffusion that individually transacts weight $x_{i,j}$ between i and j, here node i produces a single aggregate quantity X_i that accounts for all transactions between i and its neighbors. This is because the physical grid adjusts power flow according to the power flow equation (see Section 2.2) irrespective of what we prescribe. That is, each node's load, when satisfied, produces a load on the slack bus and a voltage on each node.

Thus one could say that a diffusion algorithm precisely performs the actions prescribed by the nodes in an iteration. In contrast, our algorithm only precisely computes the loads prescribed for each node. The interaction among all nodes defines the voltages. Thus the load cost prescription is somewhat followed, but possibly at the detriment of the voltage cost.

The interaction of the OMNET++ code with the load flow program is synchronous; i.e., all the nodes are synchronized after each iteration. Each node in the system has a unique ID. The communication topology and the network topology are assumed same for all of this thesis (except the "subgraph" scenario). The flowchart in Figure 4.1 shows the algorithm structure. We detail the steps below.

As described in Section 2.3 the OMNET++ environment interacts with the load flow program to simulate the grid with the DDLB algorithm deployed across it. Here we elaborate on that. Additional concepts are introduced as needed.

Initialization : The system starts from a stable state $\vec{S^0} = \langle S_i^0 \ : \ 1 \leq i \leq n\rangle$. The initialization step initializes node i to $S_i^0 = \langle V_i^0, P_i^0, Q_i^0\rangle$ where V_i^0 is the voltage,

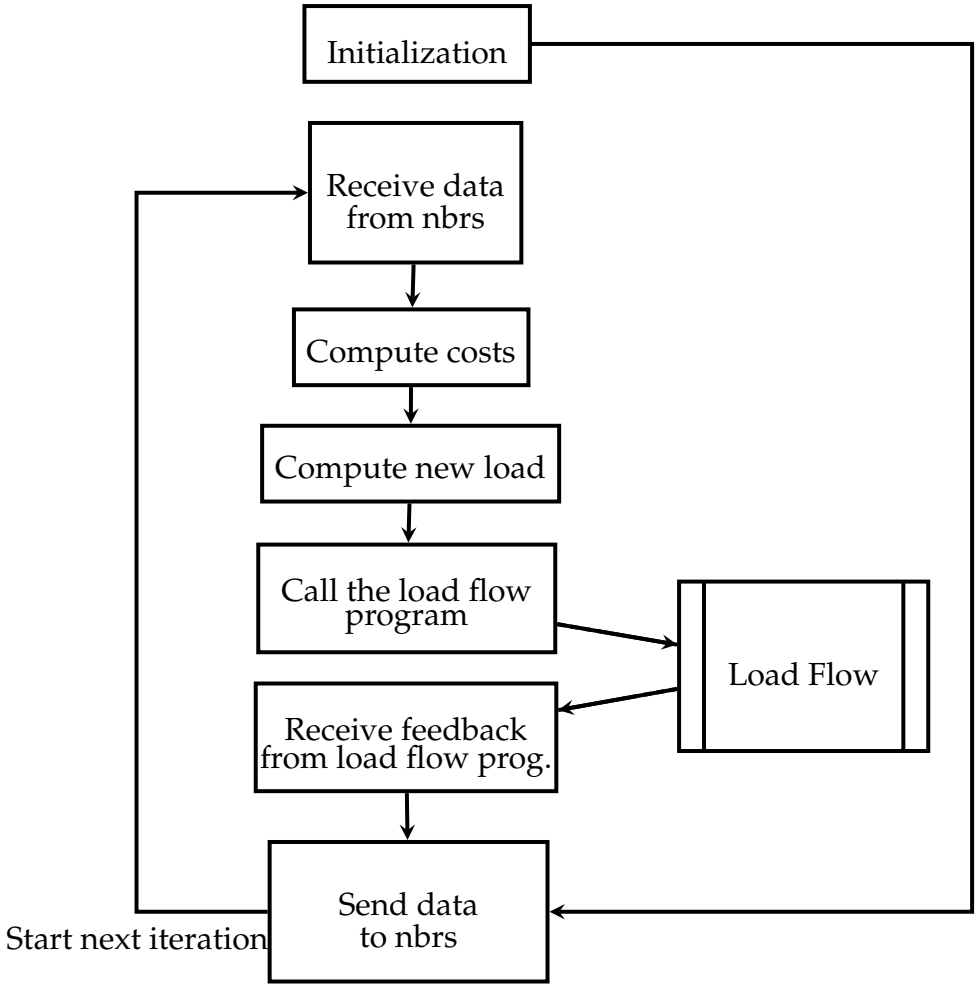


FIGURE 4.1: Algorithm Structure

P_i^0 is the active power and Q_i^0 is the reactive power for node i in the stable state. Next it uses S_i^0 to compute an initial local load cost $LC_{i,0}$ and local voltage cost $VC_{i,0}$ as detailed below.

Node profile : As part of the input to the algorithm, each node has a load and voltage profile. The *load profile* of node i is $\lambda_i = \langle \lambda_{i,-1}, \lambda_{i,0}, \lambda_{i,+1} \rangle$, where, $\lambda_{i,0}$ is the preferred load of the node. Recall that $\lambda_{i,0} > 0$ for power consumed and $\lambda_{i,0} < 0$ for power supplied. The nodes show some flexibility regarding $\lambda_{i,0}$. The range$\lambda_{i,-1} \leq \lambda_{i,0} \leq \lambda_{i,+1}$ captures this flexibility below and above $\lambda_{i,0}$.

Similarly the *voltage profile* for a node i is $\mathcal{V}_i = \langle \mathcal{V}_{i,-1}, \mathcal{V}_{i,0}, \mathcal{V}_{i,+1} \rangle$ indicates the flexibility of the voltage about the preferred point $\mathcal{V}_{i,0}$.

Load profiles can vary depending on the type of node and probably market incentives provided for flexibility. For example, the local company could provide reduced rates for customers willing to cycle the air conditioning system off for a few minutes during high load periods [13].

Voltage profiles for load nodes generally have $\mathcal{V}_{i,0} = 1$ (under the per unit system) and $\mathcal{V}_{i,-1} = 0.9$, $\mathcal{V}_{i,+1} = 1.05$ for a maximum 5% voltage variation. Source nodes generally have $\mathcal{V}_{i,0} > 1$ (around 1.03) and $\mathcal{V}_{i,-1} = \mathcal{V}_{i,0} = \mathcal{V}_{i,+1}$ (no flexibility).

Local Cost : The node profile is used to determine a local cost given the local load voltage and load. We first describe this local cost in its most general form for the load profile.

Let node i have local load profile $\lambda_i = \langle \lambda_{i,-1}, \lambda_{i,0}, \lambda_{i,+1} \rangle$. Let its current load be ℓ_i. The local load cost LC_i is a piecewise linear function illustrated in Figure 4.2. The cost function has five line segments. The central segment s_1 is between points a and c where $a = b - 0.9(b - a_1) = 0.1b + 0.9a_1$ and $c = b + 0.9(c_1 - b) = 0.9c_1 + 0.1b$. The values 0.9 and $0.1 = (1 - 0.9)$ have been chosen so that range $[a, c]$ represents 90% of the flexibility range $[a_1, c_1]$. A different number could be used.

Coming back to segment s_1, it passes through the point $(\lambda_{i,0}, 0)$ indicating that the cost for the desired load $\lambda_{i,0}$ is 0. Since the slope σ_1 of s_1, is quite small, the cost of the load in the range $[a, b]$ is also close too.

In the same spirit the cost shown by segments s_3 and s_5 is high. That is, their slopes σ_3, σ_5 are large. In between the slopes σ_2, σ_4 of the segments s_2, s_4 are more than σ_1 but less than σ_3, σ_5 to indicate that the system should be more "vigilant" as we approach the limits of the flexibility of the node.

As noted earlier, the DDLB algorithm proceeds in iterations. Initially it is to be expected that the nodes have large costs and the algorithm aims to gently nudge

the system toward a lower cost. Accordingly, σ_3 and σ_5 are somewhat small. As the algorithm proceeds, they become more aggressive as the algorithm is running out of time to take corrective action.

We set slopes $\sigma_3 = \sigma_5 = 2\sigma_2 = 2\sigma_4$. Slope σ_1 is set independently. Each of these slopes is a function of time t (iteration no.).

In general, if we wish to increase the slope from α_1 at iteration 1 to α_2 at iteration 600 (last iteration), then we set the slope at iteration t to be

$$\sigma(t) = \left(\frac{\alpha_2 - \alpha_1}{600 - 1}\right) t + \alpha_1$$

for σ_1 we have $\alpha_1 = 0.05$ and $\alpha_2 = 0.2$, and for σ_2, $\alpha_1 = 0.2$ and $\alpha_2 = 0.6$. Thus we have,

$$\sigma_1(t) = 0.00025t + 0.05$$

$$\sigma_3(t) = \sigma_5(t) = 2\sigma_2(t) = 2\sigma_4(t) = 0.00066t + 0.2$$

Thus for a piecewise continuous load cost we have,

$$LC_{i,t}(\ell) = \begin{cases} \sigma_1(t)(\ell - b), & \text{for } a \leq \ell \leq c \\ \sigma_2(t)(\ell - a) + LC_{i,t}(a), & \text{for } a_1 \leq \ell < a \\ 2\sigma_2(t)(\ell - a_1) + LC_{i,t}(a_1), & \text{for } \ell < a_1 \\ \sigma_4(t)(\ell - c) + LC_{i,t}(c), & \text{for } c < \ell \leq c_1 \\ 2\sigma_4(t)(\ell - c_1) + LC_{i,t}(c_1), & \text{for } \ell > c_1 \end{cases}$$

where, $\ell = P_{i,t-1}$.

The local voltage cost $VC_{i,t}(v)$ is also computed similarly, with points a_1, a, b, c, c_1 of loads corresponding to 0.95, 0.955, 1, 1.045, 1.05. For a generator $a_1 = a = b = c = c_1 = \mathcal{V}_{i,0}$. For the local voltage cost $VC_{i,t}$ we have, for σ_1, $\alpha_1 = 0.5$ and $\alpha_2 = 1.2$, and for σ_2, $\alpha_1 = 0.25$ and $\alpha_2 = 0.6$. Thus,

$$\sigma_1(t) = 0.00116t + 0.5$$

$$\sigma_3(t) = \sigma_5(t) = 2\sigma_2(t) = 2\sigma_4(t) = 0.00058t + 0.25$$

and for the piecewise continuous voltage cost we have,

$$VC_{i,t}(v) = \begin{cases} \sigma_1(t)(v-b), & \text{for } a \leq v \leq c \\ \sigma_2(t)(v-a) + LC_{i,t}(a), & \text{for } a_1 \leq v < a \\ 2\sigma_2(t)(v-a_1) + LC_{i,t}(a_1), & \text{for } v < a_1 \\ \sigma_4(t)(v-c) + LC_{i,t}(c), & \text{for } c < v \leq c_1 \\ 2\sigma_4(t)(v-c_1) + LC_{i,t}(c_1), & \text{for } v > c_1 \end{cases}$$

We consider four cases of the local cost corresponding to whether the load and voltage flexibility are actively used. When the load flexibility is used the cost graph has the form shown in Figure 4.2. If slopes $\sigma_1 - \sigma_5$ are all the same this flexibility is ignored and the graph has the form as shown in Figure 4.3.

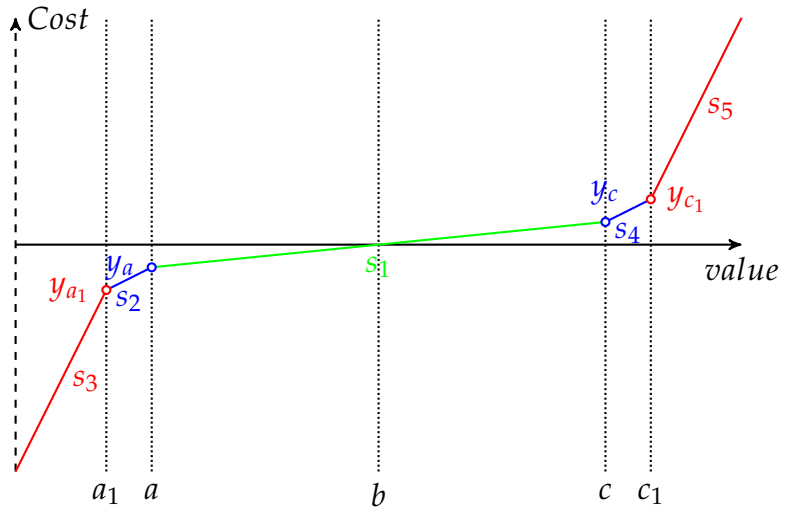


FIGURE 4.2: Local load cost, here a_1=$\lambda_{i,-1}$, b=$\lambda_{i,0}$ and $c_1 = \lambda_{i,+1}$

The four cases correspond to load and voltage flexibility used (indicated by boolean variables LF = VF = 1), neither flexibility used (LF = VF = 0), and only one of the flexibilities used (LF = 1, VF = 0 or LF = 0, VF = 1). Chapter 6 details these further.

Coming back to the algorithm, at this point each node has a local load and voltage cost. This local cost is sent to all neighbors as part of the initialization. Beyond this point we assume that we are starting an iteration and are in round t of the DDLB algorithm.

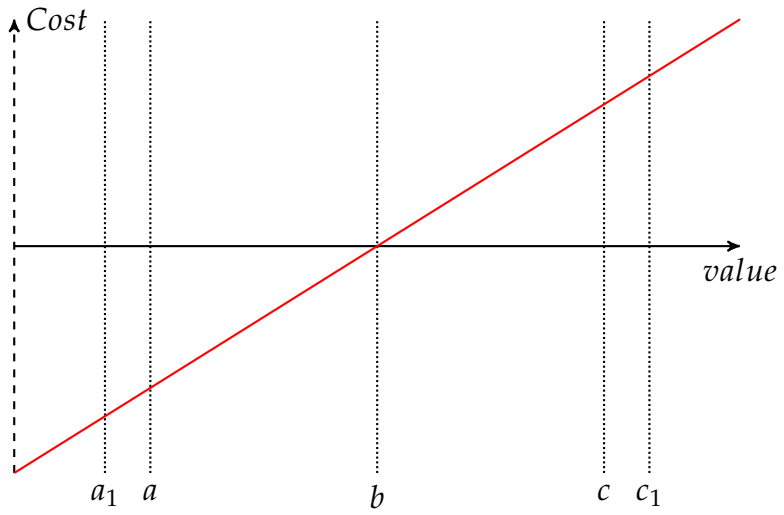


FIGURE 4.3: Local load cost, here a_1=$\lambda_{i,-1}$, b=$\lambda_{i,0}$ and $c_1 = \lambda_{i,+1}$, having same slopes $\sigma_1 \ldots \sigma_5$.

Receive data from neighbors : The local costs $LC_{j,t}$ and $VC_{j,t}$ sent from neighbors j of i are received by i which holds its own local costs $LC_{i,t}$ and $VC_{j,t}$. These quantities will be used subsequently.

Compute cost : The local "neighborhood" costs are used here to compute a "total" cost at each node. The total load cost $\mathcal{L}_{i,t}$ is computed as follows.

Let node i have degree d_i (that is, it has d_i neighbors). Define a function $\delta(d_i)$,

$$\delta(d_i) = 1 + \frac{a}{b^{d_i}} \tag{4.0.1}$$

where $a = \frac{27}{2}$ and $b = \frac{3}{2}$ (these were values selected for our program, but they could be different provided $a > b > 1$). Figure 4.4 shows the behavior of this function δ. For small d_i, $\delta(d_i)$ is large and for large d_i $\delta(d_i)$ is close to 1. Coming back to the total load cost, $\mathcal{L}_{i,t}$ we let

$$\mathcal{L}_{i,t} = \frac{\delta(d_i) LC_{i,t} + \sum_{j=0}^{d_i} LC_{j,t}}{d_i + \delta(d_i)} \tag{4.0.2}$$

Thus $\mathcal{L}_{i,t}$ is a weighted average of the local loads within the neighborhood of i. If i has very few neighbors, d_i is small and $\delta(d_i)$ is large. Therefore $L_{i,t}$ has a larger say in defining $\mathcal{L}_{i,t}$. On the other hand, if d_i is large, then $\delta(d_i) \cong 1$ and $L_{i,t}$ has not much more weight than its neighbor's local load cost. This reflects the

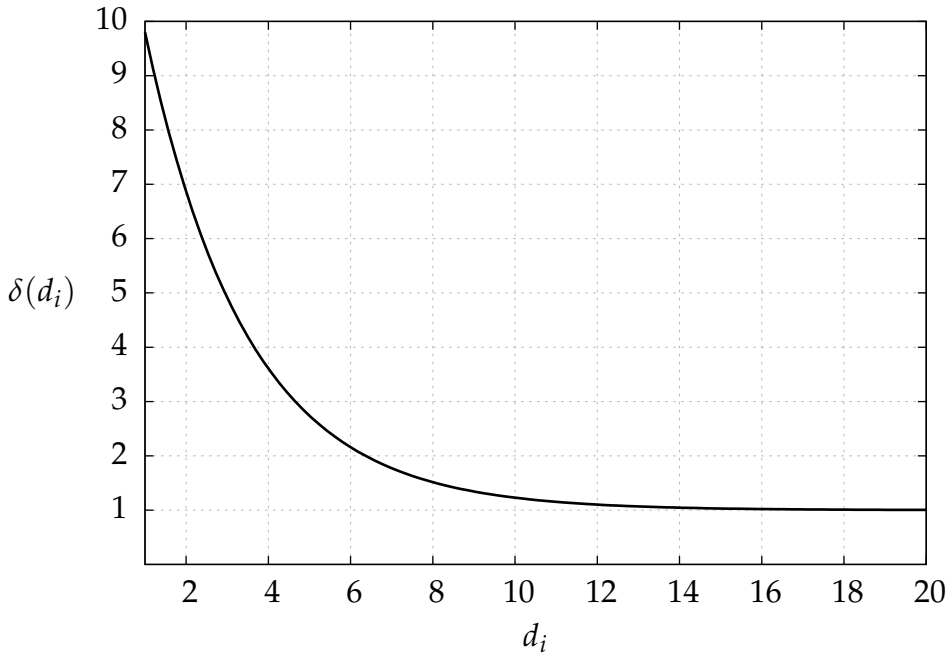


FIGURE 4.4: Behavior of the $\delta(d_i)$ function

observation that a node in an isolated situation must aggressively seek to lower its cost than one in a well-connected neighborhood. Similarly, the total voltage cost $\mathrm{V}_{i,t}$ is as in Equation (4.0.3)

$$\mathrm{V}_{i,t} = \frac{\delta(d_i) VC_{i,t} + \sum_{j=0}^{d_i} VC_{j,t}}{d_i + \delta(d_i)} \tag{4.0.3}$$

The total load cost and voltage cost are used to determine the new prescribed load for node i in iteration $t+1$. This prescription consists of a new active load $P_{i,t+1}$ and reactive load $Q_{i,t+1}$.

Compute new load : The total load cost plays a role in determining the new active load, whereas the total voltage cost is used for the reactive load. Both the costs are scaled using constants c_ℓ, c_v explained later. The new prescribed active load $\widehat{P}_{i,t+1}$ is calculated as,

$$\widehat{P}_{i,t+1} = P_{i,t} + \Delta P_{i,t} \tag{4.0.4}$$

where, $P_{i,t}$ is the present active load and $\Delta P_{i,t}$ is the change in active load calculated as follows

$$\Delta P_{i,t} = -c_\ell \left| P_{i,t} \right| \cdot \mathcal{L}_{i,t} \tag{4.0.5}$$

where $0 < c_\ell < 1$ is a load cost scaling constant. If $\mathcal{L}_{i,t}$ is positive, then the load is higher than it should be. This requires $\Delta P_{i,t}$ to be negative. Equation (4.0.5) ensures that it is. Similarly if $\mathcal{L}_{i,t} < 0$, then $\Delta P_{i,t} \geq 0$ and again Equation (4.0.5) ensures that. The constant c_ℓ makes sure that there is not too drastic a change in load (we used $c_\ell = \frac{1}{50}$). In the per unit scale $|P_{i,t}|$ is generally small ranging between -3 and 3. The value c_ℓ was experimentally determined.

The reactive load is calculated in a similar way but by using the voltage cost. Here $\widehat{Q}_{t+1}$ is the new reactive power, Q_t is the present reactive power and ΔQ is the change in reactive power and c_v is the voltage cost scaling constant.

$$\widehat{Q}_{i,t+1} = Q_{i,t} + \Delta Q_{i,t} \tag{4.0.6}$$

$$\Delta Q_{i,t} = c_v \, |Q_{i,t}| \cdot \mathrm{V}_{i,t} \tag{4.0.7}$$

Load flow program : After determining the new prescription $\langle \widehat{P}_{i,t+1}, \widehat{Q}_{i,t+1} \rangle$ at each node i, these are fed into the load flow program which simulates the action of the physical grid and returns the new system state $S_{t+1} = \langle V_{t+1}, P_{t+1}, Q_{t+1} \rangle$ where, $V_{t+1} = \langle V_{i,t+1} : 1 \leq i \leq n \rangle$, $P_{t+1} = \langle P_{i,t+1} : 1 \leq i \leq n \rangle$ and $Q_{t+1} = \langle Q_{i,t+1} : 1 \leq i \leq n \rangle$

The load flow program ensures that $P_{i,t+1} = \widehat{P}_{i,t+1}$, except if i is the slack bus.

Now $V_{i,t+1}$ is the new voltage at node i.

Send data : In the final step of the iteration each node computes local voltage and load costs $L_{i,t+1}$ and $v_{i,t+1}$ and sends them to all its neighbors as in the initialization.

The Appendix includes the OMNET++ code for the DDLB algorithm. From the discussion so far, we can see that the DDLB algorithm accepts as input, the following quantity - the n-node topology, which is specified as follows.

1. Bus information (see Section 5.1) : The number of buses and their load/voltage profiles are part of this description.

2. Line information (see Section 5.2) : The connections between the buses and their line parameters are part of this description.

5

Simulation

This chapter describes the simulation environment developed for use with this study. This environment is shown in Figure 5.1. The two key parts, OMNET++ simulator code for the DDLB algorithm and the load flow analysis program (collectively called the core code) have been detailed before (see Section 2.2 and Chapter 4). In this chapter we primarily describe the steps involved in generating the inputs, and the scripts used to run the algorithm and aggregate the data automatically.

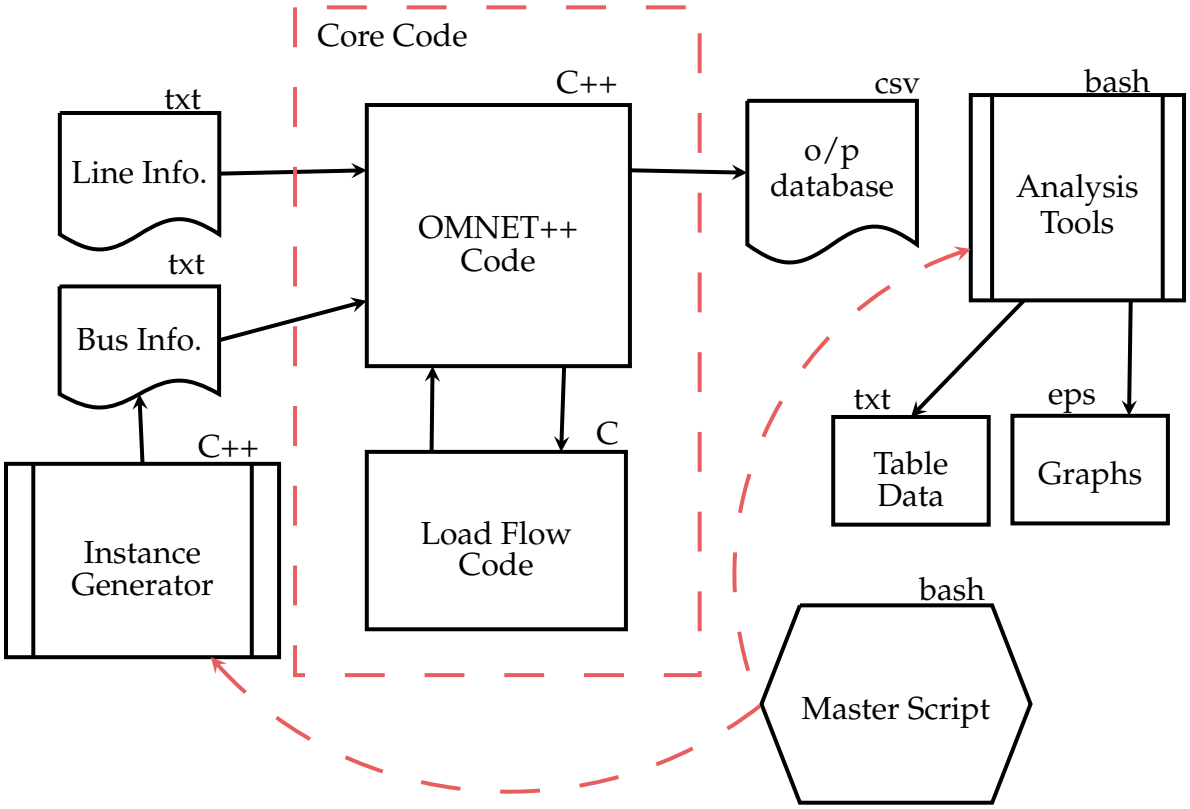


FIGURE 5.1: Simulation environment

Overall Operation: As discussed in Section 2.1 and Chapter 4, the core code requires the bus and line information to operate. The simulation environment systematically generates inputs that range across the input space and cover the scenarios of interest (balanced, overloaded, underloaded, etc.). This is done by the instance generator (a C++ script) that is explained in Section 5.3. The simulation provides a good coverage of the large input space and gives pointers to the algorithm's advantages and areas with room for improvement.

Once an input instance has been worked on by the core code, it produces a set of output files (output database) that record the data generated at each node and each iteration for that instance. This data is stored in appropriately named files as detailed in Section 5.1 and 5.2.

The Analysis tool (a bash script) can be used to access and select information for the output database and generate graphs (in eps format) and table data (in txt format).

A master script orchestrates the working of all these parts to automatically generate inputs, save output and extract data. We have used this setup for extensive simulations (ranging over many input instances such as balanced, overloaded, underloaded cases).

We now describe the components of this script architecture.

5.1 Bus information

The screenshot shown in Figure 5.2 shows the bus information matrix for a 30 node network. The columns shown in the screenshot are : node ID, nominal voltage $\mathcal{V}_0$, maximum voltage $\mathcal{V}_{+1}$, minimum voltage $\mathcal{V}_{-1}$, desired active power λ_0, maximum active power λ_{+1}, minimum active power λ_{-1}, bus type, and stable start power and voltage. Each column entry is now discussed. Node ID is a unique ID given to each node to identify it. Voltage is the desired voltage, as seen in the above example, all the buses prefer a voltage $\mathcal{V}_0 = 1$, whereas generators and the slack have higher voltages. The maximum and minimum voltages $\mathcal{V}_{+1}$, $\mathcal{V}_{-1}$ together define the voltage flexibility of the node. Column 5 is the desired active power. This value, represents a load if it is positive or a source if it is negative. Column 6 and 7 are the maximum and minimum

bus ID	voltage_(0)	voltage_(+1)	voltage_(-1)	load_(0)	load_(+1)	load_(-1)	bus_type	start_load	start_voltage
1	1.000000	1.007176	0.992824	0.530350	0.540664	0.520037	3	-0.015453	1.034544
2	1.000000	1.007176	0.992824	0.036684	0.037397	0.035970	3	0.028636	1.033290
3	1.030000	1.030000	1.030000	-0.453763	-0.444939	-0.462587	2	-0.354212	1.030000
4	1.033000	1.033000	1.033000	-0.807953	-0.792241	-0.823665	2	-0.630697	1.033000
5	1.000000	1.007176	0.992824	0.527751	0.538014	0.517488	3	-0.017483	1.033677
6	1.000000	1.007176	0.992824	0.536128	0.546554	0.525702	3	-0.010943	1.028258
7	1.030000	1.030000	1.030000	0.057096	0.058206	0.055986	2	0.044570	1.030000
8	1.030000	1.030000	1.030000	-0.443895	-0.435262	-0.452527	2	-0.346509	1.030000
9	1.000000	1.007176	0.992824	0.043346	0.044188	0.042503	3	0.033836	1.025131
10	1.000000	1.007176	0.992824	0.510595	0.520524	0.500665	3	-0.030875	1.025512
11	1.000000	1.007176	0.992824	0.047082	0.047998	0.046167	3	0.036753	1.024090
12	1.000000	1.007176	0.992824	0.057506	0.058625	0.056388	3	0.044890	1.018392
13	1.000000	1.007176	0.992824	0.005073	0.005171	0.004974	3	0.003960	1.018604
14	1.000000	1.007176	0.992824	0.501780	0.511538	0.492022	3	-0.037756	1.012442
15	1.000000	1.007176	0.992824	0.506173	0.516017	0.496330	3	0.395125	0.988165
16	1.000000	1.007176	0.992824	0.048764	0.049712	0.047816	3	0.038066	1.020890
17	1.000000	1.007176	0.992824	0.542891	0.553449	0.532334	3	-0.005664	1.024994
18	1.000000	1.007176	0.992824	0.545917	0.556534	0.535301	3	-0.003302	1.013502
19	1.000000	1.007176	0.992824	0.432609	0.441022	0.424196	3	-0.091751	1.027454
20	1.030000	1.030000	1.030000	-0.148703	-0.145811	-0.151594	2	-0.116079	1.030000
21	1.000000	1.007176	0.992824	0.054439	0.055497	0.053380	3	0.042495	1.014232
22	1.000000	1.007176	0.992824	0.360633	0.367647	0.353620	3	0.281514	1.010890
23	1.000000	1.007176	0.992824	0.472590	0.481780	0.463400	3	0.368909	0.961323
24	1.000000	1.007176	0.992824	0.112110	0.114290	0.109930	3	0.087514	0.993688
25	1.000000	1.007176	0.992824	0.068300	0.069628	0.066972	3	0.053316	1.023305
26	1.000000	1.007176	0.992824	0.545091	0.555691	0.534491	3	-0.003947	1.028754
27	1.040000	1.040000	1.040000	0.000000	0.000000	0.000000	1	0.000000	1.040000
28	1.000000	1.007176	0.992824	0.212400	0.216530	0.208269	3	0.165802	1.025390
29	1.000000	1.007176	0.992824	0.049803	0.050772	0.048835	3	0.038877	1.028325
30	1.000000	1.007176	0.992824	0.136653	0.139310	0.133995	3	0.106673	1.018709

FIGURE 5.2: Screenshot of the bus information input to the simulation

active power, which define the flexibility for the desired power. Column 8 is the bus type, which represents the different kinds of buses (nodes), slack (1), generators (2) and loads (3). Columns 9 and 10 are the stable start power and voltage respectively. This represents the state of the system at which the simulation/iteration starts. Although the last column (stable voltage) is shown, it is not part of the input. The start load and voltage represent a stable point since we want to test the stability of the system as we approach the desired value which might be either stable or unstable.

5.2 Line information

The screenshot shown in Figure 5.3 shows the line information matrix. The columns shown in the screenshot are : first two columns together form the edge (node1 , node2), for example the first line represents edge (1, 2) while the last one is for (6, 28). Columns 3-6 represent the resistance, reactance, susceptance, tap ratio and phase angle for the edge. There are 40 rows in the line information matrix, with each row representing an edge connecting two buses. Thus, this is a 30 node, 40 edge graph. Each edge is bidirectional, and resistance, reactance etc, are the parameters controlling the load flow through this edge. For our study we use only one topology with a fixed line informa-

node_a	node_b	R	X	B	tap_ratio	phase_angle
1	2	0.019200	0.057500	0.026400	1.00	0.00
1	3	0.045200	0.185200	0.020400	1.00	0.00
2	4	0.013200	0.037900	0.004200	1.00	0.00
2	5	0.047200	0.198300	0.020900	1.00	0.00
2	6	0.058100	0.176300	0.018700	1.00	0.00
4	6	0.011900	0.041400	0.004500	1.00	0.00
5	7	0.046000	0.116000	0.010200	1.00	0.00
6	7	0.026700	0.082000	0.008500	1.00	0.00
6	8	0.012000	0.042000	0.004500	1.00	0.00
6	9	0.000000	0.208000	0.004500	1.00	0.00
6	10	0.000000	0.556000	0.004500	1.00	0.00
9	11	0.000000	0.208000	0.004500	1.00	0.00
9	10	0.000000	0.110000	0.004500	1.00	0.00
4	12	0.000000	0.256000	0.004500	1.00	0.00
12	13	0.000000	0.140000	0.004500	1.00	0.00
12	14	0.123100	0.255900	0.004500	1.00	0.00
12	15	0.066200	0.130400	0.008500	1.00	0.00
12	16	0.094500	0.198700	0.018700	1.00	0.00
14	15	0.221000	0.199700	0.010200	1.00	0.00
16	17	0.082400	0.192300	0.010200	1.00	0.00
15	18	0.107000	0.218500	0.010200	1.00	0.00
18	19	0.063900	0.129200	0.010200	1.00	0.00
19	20	0.034000	0.068000	0.010200	1.00	0.00
10	20	0.093600	0.209000	0.010200	1.00	0.00
10	17	0.032400	0.084500	0.010200	1.00	0.00
10	21	0.034800	0.074900	0.010200	1.00	0.00
10	22	0.072700	0.149900	0.020900	1.00	0.00
21	22	0.011600	0.023600	0.020900	1.00	0.00
15	23	0.100000	0.202000	0.020900	1.00	0.00
22	24	0.115000	0.179000	0.021400	1.00	0.00
23	24	0.132000	0.270000	0.021400	1.00	0.00
24	25	0.188500	0.329200	0.021400	1.00	0.00
25	26	0.254400	0.380000	0.021400	1.00	0.00
25	27	0.109300	0.208700	0.021400	1.00	0.00
27	28	0.000000	0.396000	0.021400	1.00	0.00
27	29	0.219800	0.415300	0.021400	1.00	0.00
27	30	0.320200	0.602700	0.021400	1.00	0.00
29	30	0.239900	0.453300	0.021400	1.00	0.00
8	28	0.063600	0.200000	0.021400	1.00	0.00
6	28	0.016900	0.059900	0.006500	1.00	0.00

FIGURE 5.3: Screenshot of the line information input to the simulation

tion. However, in principle this could be altered as required. The topology used was the IEEE 30-bus network [9].

5.3 Instance Generation

To understand how the input instances to the core code are generated, we have to first introduce the idea of input state which is simply an instance of the line and bus information. This means that the entire network's parameters at a given time are represented through this state. Thus the network state can be viewed as either an input to the core program or its output. Further, since we do not change the information, we seek to look at variations only in the bus information.

There can be different types of states. One way to classify them is as either stable or unstable. A *stable state* of the network is one in which voltages for all nodes are within the acceptable range. Even if a single node has voltage outside the acceptable range

$[\mathcal{V}_{-1}, \mathcal{V}_{+1}]$, then the network state is unstable. A state is *convergent* if the load flow program converges for that state and produces a "non-NaN" value for each load.

A network state is *balanced* iff $\sum_{i=1}^{n-1} \ell_i = 0$, where ℓ_i is the load of node i; we have assumed node n to be the slack bus for this discussion. In a balanced state the power generated is equal to the power consumed.

A base state is a stable, balanced state. Base states are used in our simulation to generate several comparable, yet distinct scenarios, and as the stable starting point for these scenarios.

Starting from a primary base state $\vec{S}_p$ we will generate secondary base states $\vec{S}_{p,s}$. From each of these secondary base states $\vec{S}_{p,s}$ we will generate a set of states (input instances for the core program). Each input state will be applied to graph $\mathcal{G}$, the 30-node 40-link IEEE standard system and to $\widehat{\mathcal{G}}$ a subgraph of $\mathcal{G}$ with only $27(66\%)$ of the edges of $\mathcal{G}$. Edges of $\mathcal{G}$ are deleted on the basis of their line impedences (large impedences indicating longer edges). Specifically the longest 33% of edges (13 in our case) are removed to generate $\widehat{\mathcal{G}}$. We will explain the instance generation through pseudo-codes given below.

Procedure Generate_P_Base() shown in Figure 5.5 is used to generate a primary base state $\vec{S}$, given a primary base state $\vec{S}$. (Initially a primary base state is generated manually to start the procedure off). This procedure is called to generate 10000 primary base states.

The input primary base state $\vec{S} = \langle \vec{P}, \vec{Q}, \vec{V} \rangle$ (active load, reactive load, voltage vectors across the nodes) is first permuted to generate $\vec{S}'$; That is, the node id's of $\vec{S}$ are permuted. Specifically, let $V = \{1, 2, \ldots, n\}$ be the set of nodes and let $f : V \to V$ be a bijection. The effect of permuting the node id's is best illustrated by the example in Figure 5.4. For this example $V = \{1, 2, \ldots, 8\}$. Let $f(i) = i + 1$ for $1 \leq i < 8$ and let $f(8) = 1$.

Figure 5.4 (b) shows the permuted topology. Note that the bus information for node i is the same. That is, for example, node 7 is still the slack bus, but is now

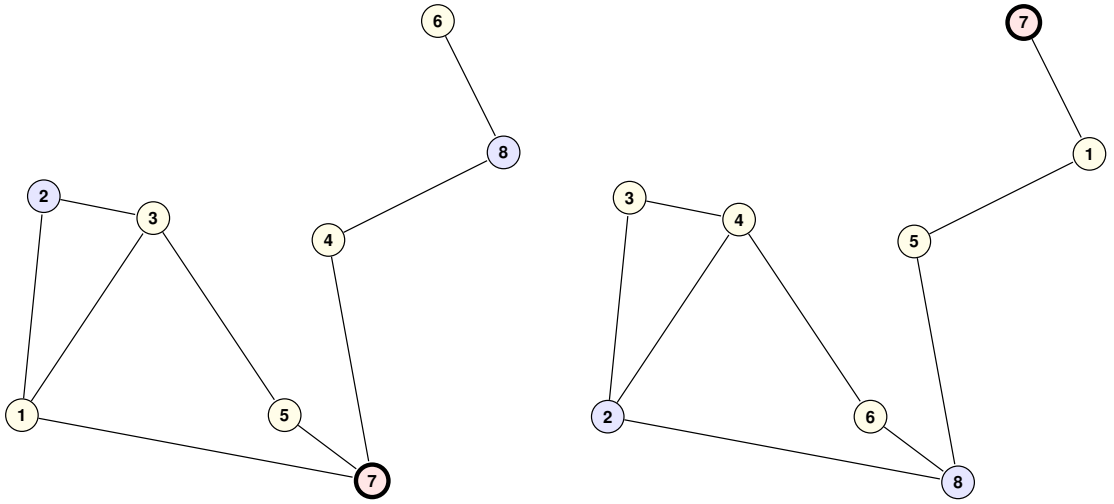


FIGURE 5.4: An example physical power grid (a) and its graph representation (b)

topologically in a different part of the graph. The line information for original line (i, j) now is the information for line $(f(i), f(j))$.

Thus the effect of the permutation is to shuffle the positions of nodes in the topology. While the two topologies are isomorphic the node properties render the underlying grids very different. For example, in the first case the generators 2 and 8 are not adjacent, as they are in the permuted case. This can affect the behavior of the grid. The loads in the permuted state $\vec{S'}$ are next changed as shown in Figure 5.5. If the state $\vec{S'}$ has $P_i > 0$, then $-2P_i \leq R_i \leq 2P_i$. The new value of P_i is $P_i(new) = P_i + R_i$ which satisfies $-P_i \leq P_i(new) \leq 3P_i$. Thus if $P_i > 0$ then $P_i(new)$ is more likely to be positive. Similarly for negative P_i, we have $-3P_i \leq P_i(new) \leq P_i$. In either case, P_i tends to keep the same characteristics as that of P_i from the permutation. But the loads are different now.

The new state with new loads is denoted by $\vec{S''}$. We now normalize these loads to get a balanced state, by setting the sum of the loads of all the states to 0. The slack bus already has as 0 load (for the input) and does not affect the average. The resulting state is $\vec{S_p}$. This is an acceptable (balanced) primary state if it is stable.

```
Generate_P_Base (S⃗=⟨P⃗, Q⃗, V⃗⟩)
/*Generate Primary base state S⃗_p*/
  repeat
    S⃗′=⟨P⃗′, Q⃗′, V⃗′⟩ with nodes of S⃗ randomly permuted
    /* permuted state S⃗″ */
    for each node i of S⃗′ with active load P_i′ do
      r_i ← rand[1,2] /* 1 ≤ r_i ≤ 2 is a random number */
      Temp_i ← P_i′ * rand[−r_i, r_i]
      S⃗″=⟨P⃗″, Q⃗″, V⃗″⟩ is S⃗′ with P_i′ replaced by Temp_i
    end     /* loads permuted for S⃗″ */
    a ← (1/n) Σ_{i=1}^{n} P_i″ average active load of S⃗″
    for each node i of S⃗″ with active load P_i″ do
      Temp_i ← P_i″ − a
      S⃗_p = ⟨P⃗_p, Q⃗_p, V⃗_p⟩ is S⃗″ with P_i″ replaced by Temp_i
    end   until S⃗_p is stable
  return resulting S⃗_p
end
```

FIGURE 5.5: Algorithm for primary base state

Procedure Generate_S_Base() shown in Figure 5.6 takes a primary base state $\vec{S}_p$ obtained from the Generate_P_Base() and generates a secondary base state $\vec{S}_{p,s}$, where $1 \leq v \leq 10$. As is clear from the procedure, each secondary base $\vec{S}_{p,s}$ is a stable permutation of the primary base $\vec{S}_p$

```
Generate_S_Base(S⃗_p=⟨P⃗, Q⃗, V⃗⟩)
/*Generate secondary base state S⃗_{p,s}*/
  repeat
    S⃗_{p,s} = ⟨P⃗′, Q⃗′, V⃗′⟩ with nodes of S⃗_p randomly permuted
  until S⃗_{p,s} is stable
  return S⃗_{p,s}
end
```

FIGURE 5.6: Algorithm for secondary base state

Procedure Generate_Balanced() (see Figure 5.7) takes a secondary base state $\vec{S}_{u,v}$ obtained from the Generate_S_Base and randomly generates a balanced state. The main action of the procedure is to randomly change loads while keeping the state balanced. The last two lines indicate that the state is to be run once on graph $\mathcal{G}$ and once on $\widehat{\mathcal{G}}$ (the same grid with reduced communication edges). Sim-

```
Generate_Balanced(S⃗_{p,s}=⟨P⃗, Q⃗, V⃗⟩)
/* Generate a balanced state S⃗_{p,s,b} */
  repeat
    r ← rand[0.96, 1.44]
    S⃗_{p,s,b}=⟨P⃗′, Q⃗′, V⃗′⟩ with P′_i ← P_i * r
  until S⃗_{p,s,b} is convergent
  save instance ⟨S⃗_{p,s,b}, 𝒢⟩
  save instance ⟨S⃗_{p,s,b}, 𝒢̂⟩
end
```

FIGURE 5.7: Algorithm for generation of a balanced state

ilarly Generate_Overloaded() and Generate_Underloaded() (see Figure 5.8 and 5.9) each generate one overloaded and one underloaded state respectively. Both these scenarios are created either by increasing loads (*loads*), decreasing generation (*gens*), or both (*both*). Amount of load increased is regulated by r_1, r_2. All these states (balanced, overloaded, underloaded) are generated from the same secondary base state.

Edges of $\mathcal{G}$ are deleted on the basis of their line impedences (large impedences indicating longer edges). Specifically the longest 33% of edges (13 in our case) are removed to generate $\widehat{\mathcal{G}}$.

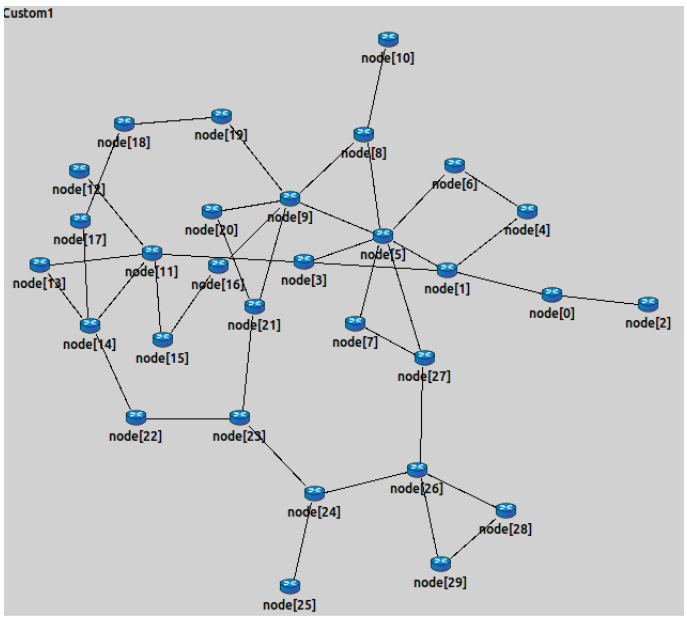


FIGURE 5.10: Screenshot of the the full network topology

```
Generate_Overload(S⃗_{p,s}=⟨P⃗, Q⃗, V⃗⟩)
/* Generate overloaded state S⃗_{p,s,o} */
  repeat
    r_1 ← rand{0.1, 0.2, 0.3}
    /* randomly pick one value from set {0.1, 0.2, 0.3} */
    r_2 ← rand{loads, gens, both}
    /* perturb loads, generators or both */
    r_3 ← rand[0.84, 1.5]
    Abs ← Σ_{i=1}^{n} |P_i|
    G ← no. of generators in S⃗_{p,s}
    L ← (n − 1) − G      /* no. of non-slack loads in S⃗_{p,s} */
    Case (r_2)
      loads : for each load i do P'_i ← P_i + r_1 * Abs/L
      gens : for each gen i do P'_i ← P_i + r_1 * Abs/G
      both : for all nodes i do P'_i ← P_i + r_1 * Abs/(L + G)
    end
    for each node i do P'_i ← r_3 * P'_i
    S⃗_{p,s,o} ← ⟨P⃗', Q⃗, V⃗⟩
  until S⃗_{p,s,o} is convergent
  save instance ⟨S⃗_{p,s,o}, G⟩
  save instance ⟨S⃗_{p,s,o}, Ĝ⟩
end
```

FIGURE 5.8: Algorithm for generation of an overloaded state

Figure 5.10 shows the original IEEE 30-bus topology with 40-edges and Figure 5.11 shows the subgraph with 27 edges. As observed before, the subgraph will be used to see the effect of restricted communication on the DDLB algorithm. We use only the 27-edge of Figure 5.11 for our study.

Procedure Generate_Radial() (see Figure 5.12) uses a balanced state $\vec{S}_{p,s,b}$ to produce potentially overloaded states reflective of local generator damage. Figure 5.12 shows our approach. The procedure uses a parameter d, representing the "radius" of the damage. The basic idea is to pick a random node i and set the loads of all generator nodes at distance $\leq d$ from i to 0.

Procedure Generate_Input_Instances (see Figure 5.13) is the systematic approach used to generate the various instances using the procedures described earlier. In total 1.6 million input instances are generated as explained below. The algorithm gen-

```
Generate_Underload(S⃗_{p,s}=⟨P⃗, Q⃗, V⃗⟩)
/* Generate underloaded case S⃗_{p,s,u} */
  repeat
    r_1 ← rand{−0.1, −0.2, −0.3}
  /* randomly pick one value from set {−0.1, −0.2, −0.3} */
    r_2 ← rand{loads, gens, both}
    /* perturb loads, generators or both */
    r_3 ← rand[0.84, 1.5]
    Abs ← Σ_{i=1}^{n} |P_i|
    G ← no. of generators in S⃗_{p,s}
    L ← (n − 1) − G      /* no. of non-slack loads in S⃗_{p,s} */
    Case (r_2)
      loads : for each load i do P'_i ← P_i + r_1 * Abs/L
      gens : for each gen i do P'_i ← P_i + r_1 * Abs/G
      both : for all nodes i do P'_i ← P_i + r_1 * Abs/(L + G)
    end
    for each node i do P'_i ← r_3 * P'_i
    S⃗_{p,s,u} ← ⟨P⃗', Q⃗, V⃗⟩
  until S⃗_{p,s,u} is
  save instance ⟨S⃗_{p,s,u}, G⟩
  save instance ⟨S⃗_{p,s,u}, Ĝ⟩
end
```

FIGURE 5.9: Algorithm for generation of an underloaded state

erates 10000 primary base states. For each of these there are 10 secondary base states. Each secondary base state produces one each of balanced, overloaded, and underloaded states and 5 each of radial 1 and radial 2 states. Each of these balanced, overloaded, and underloaded states is also run on the subgraph $\widehat{\mathcal{G}}$. Table 5.1 shows the number of inputs tested for the various scenarios.

TABLE 5.1: No. of inputs for each scenario

Scenario	No. of inputs generated
Balanced	100,000
Overloaded	100,000
Underloaded	100,000
Radial 1	500,000
Radial 2	500,000
Subgraph	300,000
Overall	1,600,000

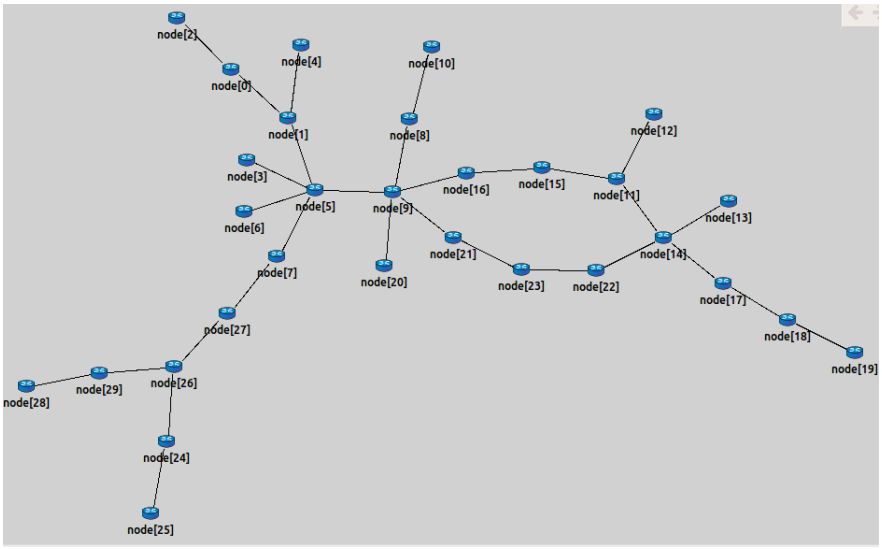


FIGURE 5.11: Screenshot of the subgraph scenario topology

```
Generate_Radial(S⃗_{p,s,b}, d) /*d ← {1,2}*/
  for r ← 1 to 5 do
    repeat
      pick random node i of S⃗_{p,s,b}
      for each source j within distance d of i
        set P_j ← 0
        S⃗^{d,r}_{p,s,b} is S⃗_{p,s,b} with above P_j's set to 0      end
    until S⃗^{d,r}_{p,s,b} is stable
    save S⃗^{d,r}_{p,s,b} in List
  end
  return List
end
```

FIGURE 5.12: Algorithm for radial failure states

5.4 Additional Tools

Referring back to Figure 5.1 we see that we are yet to discuss the following blocks - output database, analysis tool, table data, graphs, and the master script. Each run of our algorithm produces data, which needs to be properly organized, so that we can compare and analyze results. This is done by properly naming each output file and placing them into organized folders, which collectively forms the *output database*. The *Analysis tool* is a set of bash shell scripts written for conveniently converting the raw data from the output database into meaningful tables and graphs. The *Master Script* is the backbone of the entire simulation, which is also a bash script written to

```
Generate_Input_Instances
  Manually generate primary base S⃗
  for p ← 1 to 10000 do
    S⃗_p ← Generate_P_Base(S⃗)
    for s ← 1 to 10 do
      S⃗_{p,s} ← Generate_S_Base(S⃗_p)
      S⃗_{p,s,b} ← Generate_Balanced(S⃗_{p,s})
      S⃗_{p,s,o} ← Generate_overload (S⃗_{p,s})
      S⃗_{p,s,u} ← Generate_underload (S⃗_{p,s})
      R^1_{p,s,b}[1..5] ← Generate_radial(S⃗_{p,s,b}, 1)
      R^2_{p,s,b}[1..5] ← Generate_radial(S⃗_{p,s,b}, 2)
    end
    S⃗ ← S⃗_p
  end
end
```

FIGURE 5.13: Algorithm for systematically generating all input instances

easily automate the entire process of input instance creation, simulation run, storing of output files, conversion into tables and graphs.

6

Simulation Results

In Chapter 5 we described how inputs for different grid scenarios are generated. We also detailed the number of inputs generated per scenario, and discussed how these inputs cover a wide range of instances over the problem space. In this chapter the results of our simulation are discussed.

For each input, the DDLB algorithm iteratively runs for 600 iterations. After several trials, we decided on 600 iterations, as it seemed large enough for most cases to coverage to a stable output.

As detailed in Chapter 2, the one-shot algorithm applies desired loads to the grid (here the load flow program) directly without any concern for the voltage stability. It is the simplest (brute-force) distributed load balancing algorithm. We compare the performance of the DDLB algorithm to the one-shot algorithm. The above ideas of input convergence, voltage, load and slack bus costs (described in Sections 6.1– 6.6) all apply to the one-shot algorithm as well as the DDLB algorithm. The reader should also recall the 6 scenarios described in Chapter 5 : balanced, overloaded, underloaded, radial 1, radial 2 and subgraph.

In each of the following sections, we first describe the main considerations that we discuss through our results. These include input and output convergence, voltage, load and slack bus errors, and algorithm types (one-shot and flexibility variations).

6.1 Input Convergence

Recall that the DDLB algorithm calls the load flow program at each iteration. The load flow program, in turn, iteratively solves the power flow equations (see Equations (2.2.1) and (2.2.2)). If the load flow program is unable to converge for the given input,

then it does not produce a valid output. As a result, the DDLB algorithm also does not produce a valid output for its given input. Thus we will say that an input (to the DDLB algorithm) *converges* iff for each of its 600 iterations, the call to the load flow program converges. As shown in Table 6.1 most inputs to the DDLB and one-shot algorithms converge. Table 6.1 shows the input convergence for different scenarios. It is clear that only a small fraction of the 1.6 million inputs tested were rendered unusable.

TABLE 6.1: Input Convergence failure (%)

Scenario	No. of simulated inputs	Only DDLB (%)	Only One-Shot(%)	Both DDLB and One-Shot(%)
Balanced	100000	0.00	0.64	0.02
Overloaded	100000	1.56	2.97	5.11
Underloaded	100000	0.03	0.14	0.01
Radial 1	500000	1.45	2.41	4.52
Radial 2	500000	1.72	3.62	5.37
Subgraph	300000	0.59	0.74	0.31
Overall	1600000	1.25	2.56	4.21

6.2 Output Convergence

For DDLB inputs that converge, the algorithm slowly nudges the grid towards a stable output. For any iteration $1 \leq t \leq 600$, let X_t represent the output of the DDLB algorithm after iteration t. Here output X may be, for example, the average voltage across all nodes, or the load error (defined later) etc. We say that output X converges in iteration t_o iff for any $t_o \leq t \leq 600$,

$$\frac{|X_t - X_{600}|}{Max(\epsilon, X_{600})} \leq 5\% \tag{6.2.1}$$

For quantities that have $X_{600} \to 0$, we use a minimum value ϵ for X_{600} to allow for output convergence to be measured without divide-by-0 error.

Table 6.2 shows the load convergence[1] of the DDLB algorithm for the various scenarios. It is clear that balanced instances (including for subgraphs) require the fewest iterations to converge. Localized problems (radial damage) cause the most disruption to the output convergence. Figure 6.1 shows the histogram of load convergence for

[1] where X_t is the average active load at the end of iteration t across all 29 buses(excluding the slack bus) of the grid

TABLE 6.2: No. of iterations for load convergence

Scenario	Load Convergence (iteration no.)
Balanced	42
Overloaded	213
Underloaded	223
Radial 1	309
Radial 2	489
Subgraph (Balanced)	40
Subgraph (Overloaded)	200
Subgraph (Underloaded)	205
Overall	219

balanced, underloaded and overloaded scenarios, each of which has 100,000 simulation inputs. The histogram indicated that the averages of Table 6.2 are given indicative of the convergence behaviors that we do not have a large spread of the convergence iterations. For example, the balance case with an average convergence of 42 iterations, virtually has no instances after 100 iterations. The overloaded and underloaded cases show similar properties. The few stray cases in iterations 550–600 had very small numbers for which a small value of ϵ in Equation 6.2.1 may have worked better.

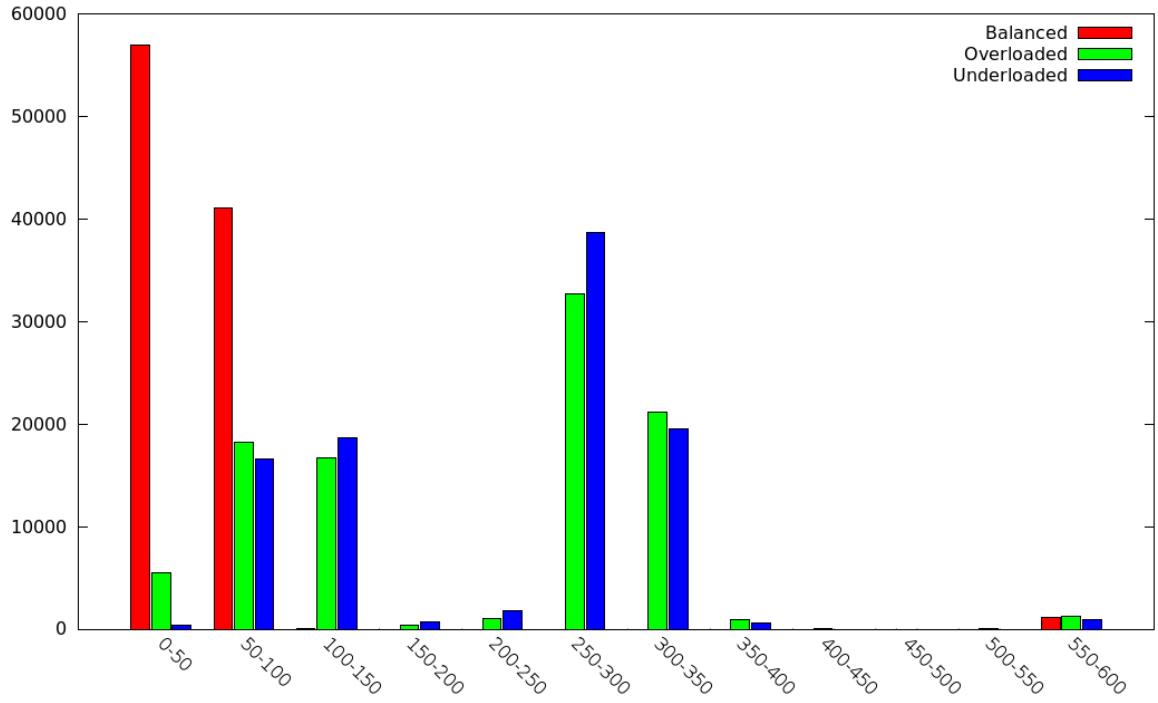


FIGURE 6.1: Histogram of load convergence for balanced, underloaded and overloaded scenarios

6.3 Voltage Error

We use two interpretations of where a poor performance of the algorithm expresses itself through voltage. Recall that each node i has a voltage profile $\mathcal{V}_i = \langle \mathcal{V}_{i,-1}, \mathcal{V}_{i,0}, \mathcal{V}_{i,+1} \rangle$ where $\mathcal{V}_{i,0}$ is the desired voltage and $[\mathcal{V}_{i,-1}, \mathcal{V}_{i,+1}]$ is the range of acceptable voltages. Let V_i be the actual voltage of node i at the iteration when the voltage error is being measured. The first voltage error ($VE1$), a binary quantity, is defined as follows.

$$VE1 = 1, \quad \text{iff } \forall\ 1 \leq i \leq n, \mathcal{V}_{i,-1} \leq V_i \leq \mathcal{V}_{i,+1} \tag{6.3.1}$$

This error measures whether the overall system is stable ($VE1 = 1$) or not ($VE1 = 0$). The second voltage error (VE2) measures the deviation of the voltage from the desired value $\mathcal{V}_{i,0}$.

$$VE2 = \frac{|V_i - \mathcal{V}_{i,0}|}{\mathcal{V}_{i,0}} \tag{6.3.2}$$

Here $\mathcal{V}_{i,0} \geq 1$ so a divide-by-0 error is not possible. Table 6.3 shows $VE1$ and $VE2$ for various scenarios. It is clear that the DDLB algorithm gives better results for voltage error than the one-shot algorithm. Even in the extreme cases like overloaded and radial 2, only 1.46% and 0.99% of the cases fail respectively ($VE1$). Compare this to the 57% failure rate of the one-shot algorithm across all the cases.

The DDLB algorithm has $VE2 < 1.19\%$ for all cases which is extremely low compared to the lowest $VE2$ of 6.77% of the one-shot algorithm in the case of underloaded scenario. Figures 6.2 shows how $VE2$ changes across the 600 iterations of the DDLB algorithm. For clarity, the last 100 iterations are shown in Figure 6.3.

6.4 Load Error

As in the case of voltage, we have two load errors. Recall that the load profile of node i is $\lambda_i = \langle \lambda_{i,-1}, \lambda_{i,0}, \lambda_{i,+1} \rangle$ where $\lambda_{i,0}$ is the preferred load and $[\lambda_{i,-1}, \lambda_{i,+1}]$ expresses its flexibility. Let P_i be the active load of node i at the point the error is being measured.

TABLE 6.3: Voltage error [%]

Scenario	Average $VE1$		Average $VE2$	
	DDLB	One-Shot	DDLB	One-Shot
Balanced	0.12	45.95	0.47	7.89
Overloaded	1.46	57.90	0.55	14.66
Underloaded	0.27	58.43	0.47	6.77
Radial 1	0.42	56.44	0.75	15.95
Radial 2	0.99	59.36	1.19	22.14
Subgraph (Balanced)	0.11	45.95	0.43	7.89
Subgraph (Overloaded)	1.12	57.90	0.49	14.66
Subgraph (Underloaded)	0.24	58.43	0.44	6.77
Overall	0.63	57.12	0.52	14.45

The first load error ($LE1$) is defined as follows,

$$LE1 = \begin{cases} \dfrac{P_i - \lambda_{i,+1}}{\lambda_{i,0}}, & \text{if } P_i > \lambda_{i,+1} \\ \dfrac{\lambda_{i,-1} - P_i}{\lambda_{i,0}}, & \text{if } P_i < \lambda_{i,-1} \\ 0, & \text{if } \lambda_{i,+1} \leq P_i \leq \lambda_{i,+1} \end{cases}$$

This accounts for the fact that the node shows some flexibility; ϵ accounts for divide-by-zero errors. The second load error ($LE2$) is defined as follows,

$$LE2 = \frac{P_i - \lambda_{i,0}}{Max(\epsilon, \lambda_{i,0})} \tag{6.4.1}$$

This measures the deviation from the desired load $\lambda_{i,0}$. The Table 6.4 shows $LE1$ and $LE2$ for various scenarios.

TABLE 6.4: Load error [%]

Scenario	Average $LE1$		Average $LE2$	
	DDLB	One-Shot	DDLB	One-Shot
Balanced	0.00	0	0.02	0
Overloaded	15.08	0	21.59	0
Underloaded	0.41	0	0.42	0
Radial 1	0.70	0	0.89	0
Radial 2	1.71	0	2.14	0
Subgraph (Balanced)	0.00	0	0.04	0
Subgraph (Overloaded)	13.30	0	17.32	0
Subgraph (Underloaded)	0.39	0	0.40	0
Overall	1.23	0	1.52	0

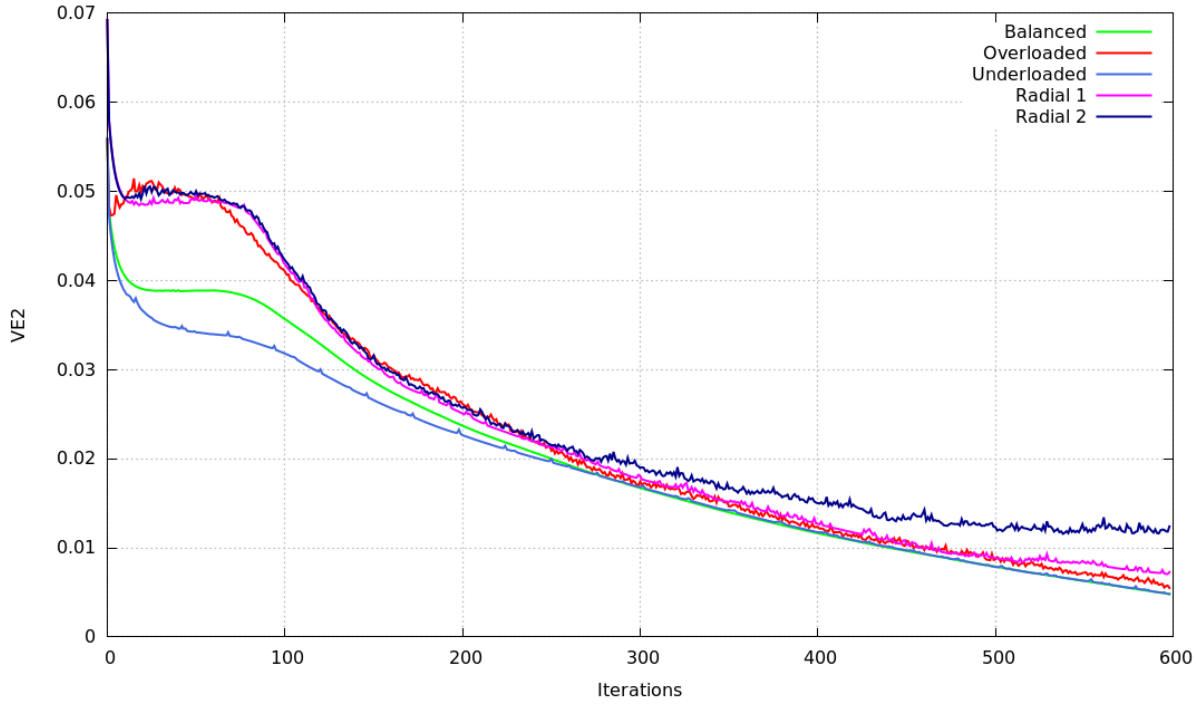


FIGURE 6.2: Average $VE2$ for various scenarios of the DDLB algorithm execution

From Table 6.4 we observe that the maximum load error $LE1$ is for the overloaded case 15%, while the balanced scenario has almost zero error. These results indicate that with a small load flexibility, the voltage error can be brought down significantly (compare the values for VE2 in Table 6.3, overloaded VE2 is only 0.55% for DDLB, compared to 14.66% for one-shot). Figures 6.4, 6.5, 6.6 and 6.7 show $LE1$ for various scenarios of running the DDLB algorithm.

6.5 Slack Bus Error

The role of the slack bus is to cover for any imbalance between the source and load powers (and losses in the grid). Recall that the active power consumed by node i at the time this error is being measured is P_i. Let P_s be the power at the slack bus. Let $\mathbb{N} = \{1, 2, \ldots, n\} - \{\text{Slack bus}\}$. Let $\mathbb{L} \subseteq \mathbb{N}$ be the set of nodes (buses) with $P_i \geq 0$; that is they consume power. Let $\mathbb{S} = \mathbb{N} - \mathbb{L}$ be the set of sources. Ideally the slack power $P_s = \sum_i P_i$ (no counting power losses). The slack error is defined as

$$SE = \frac{P_s - \sum\limits_{i \in \mathbb{N}} P_i}{2Max\left(\sum\limits_{i \in \mathbb{L}} P_i, \sum\limits_{i \in \mathbb{S}} P_i\right)} \tag{6.5.1}$$

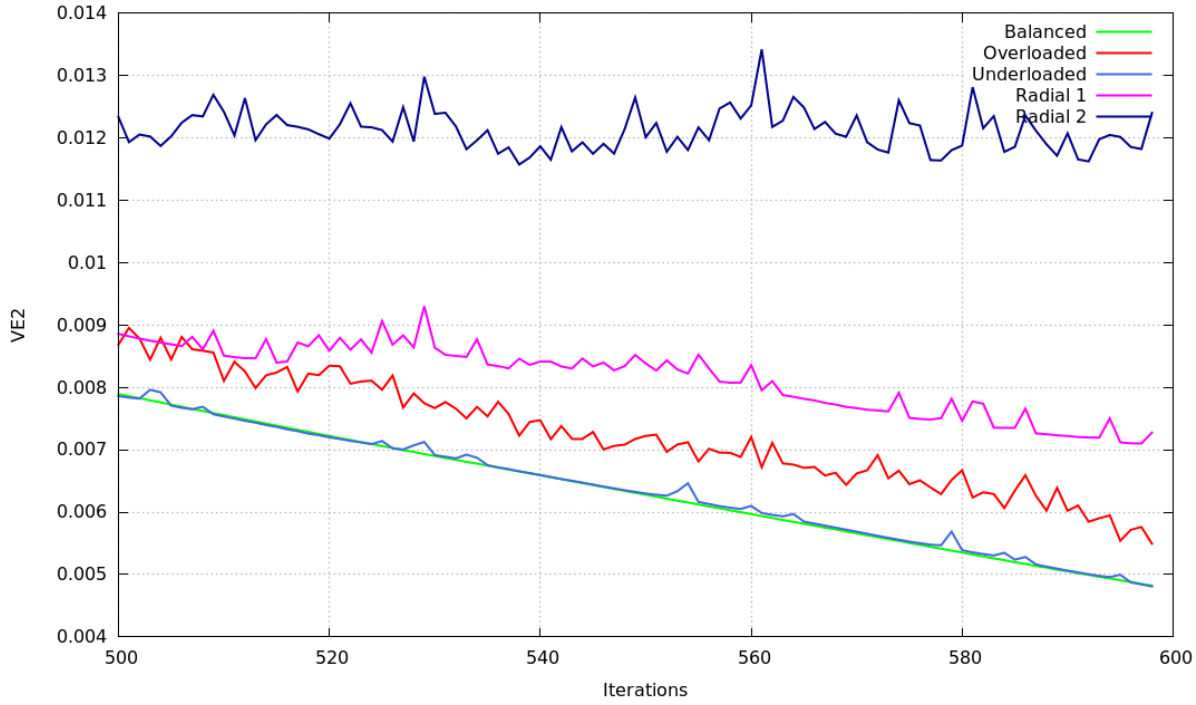


FIGURE 6.3: Average *VE*2 for various scenarios of the DDLB algorithm execution for iteration 500-600

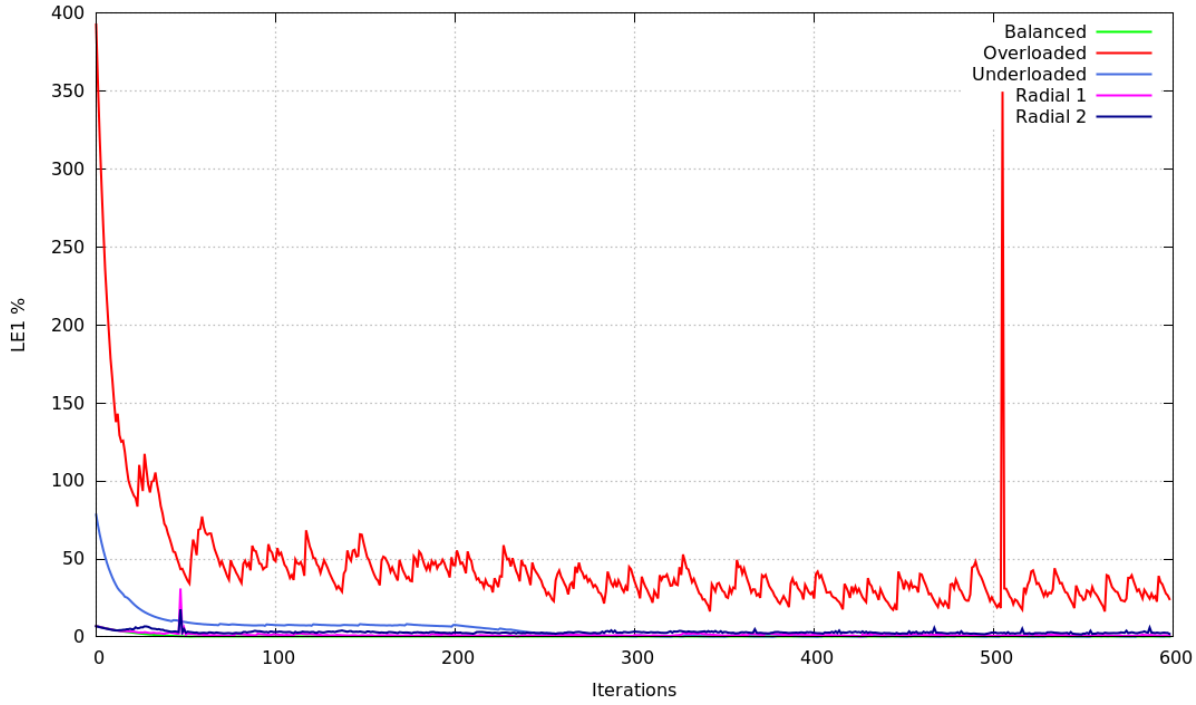


FIGURE 6.4: Average *LE*1 for various scenarios of the DDLB algorithm execution

Table 6.5 shows slack error for various scenarios. It is clear from the observations in Table 6.5 that the slack error is not significantly different for the DDLB and one-shot algorithms. For most cases the DDLB algorithm has a larger slack error. However, this

TABLE 6.5: Slack error [%]

Scenario	DDLB	One-Shot
Balanced	93	94
Overloaded	80	69
Underloaded	77	77
Radial 1	90	90
Radial 2	85	80
Subgraph (Balanced)	92	94
Subgraph (Overloaded)	79	69
Subgraph (Underloaded)	75	77
Overall	84	83

error is a small price to pay for the vastly improved voltage stability due to the DDLB algorithm.

6.6 Flexibility Variations

As detailed in Chapter 4 the DDLB algorithm uses a piecewise linear function to capture the cost of loads and voltages, consistent with the flexibilities of their profiles. The algorithm uses these local costs to progress towards the solution. By setting $\lambda_{i,-1} = \lambda_{i,0} = \lambda_{i,+1}$ or $\mathcal{V}_{i,-1} = \mathcal{V}_{i,0} = \mathcal{V}_{i,+1}$, the flexibility can be eliminated (or ignored) in the corresponding local cost function. We consider the four combinations

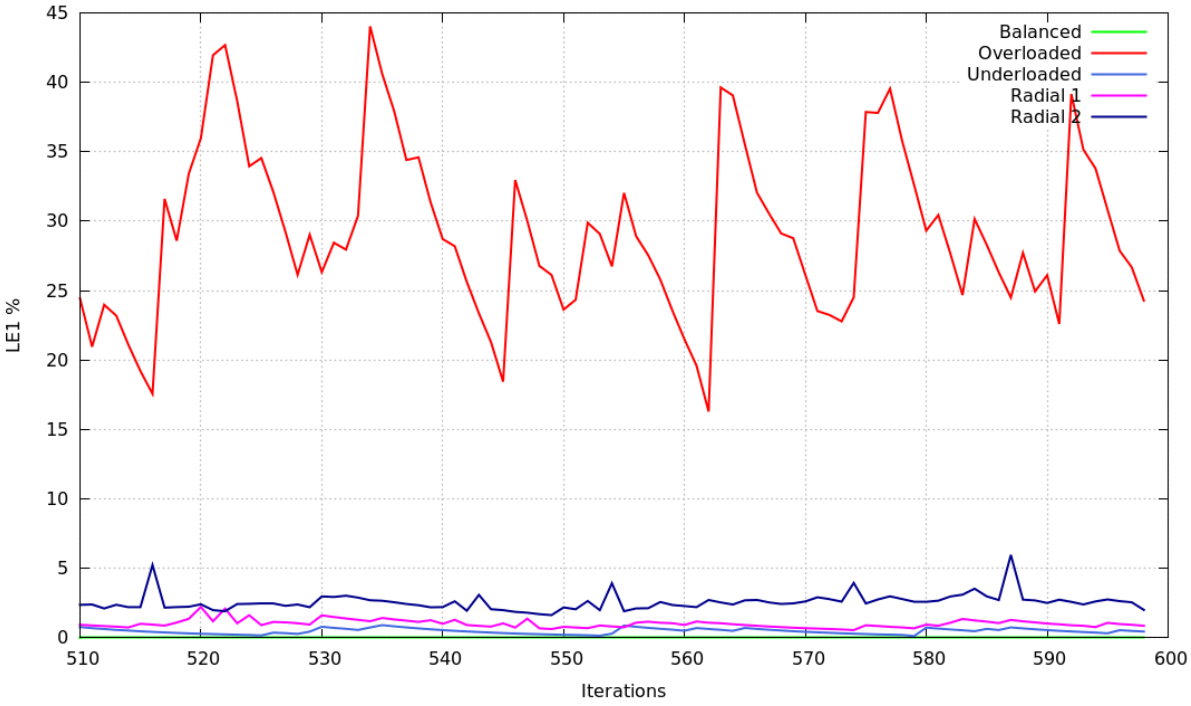


FIGURE 6.5: Average $LE1$ for various scenarios of the DDLB algorithm execution for iteration 510–600

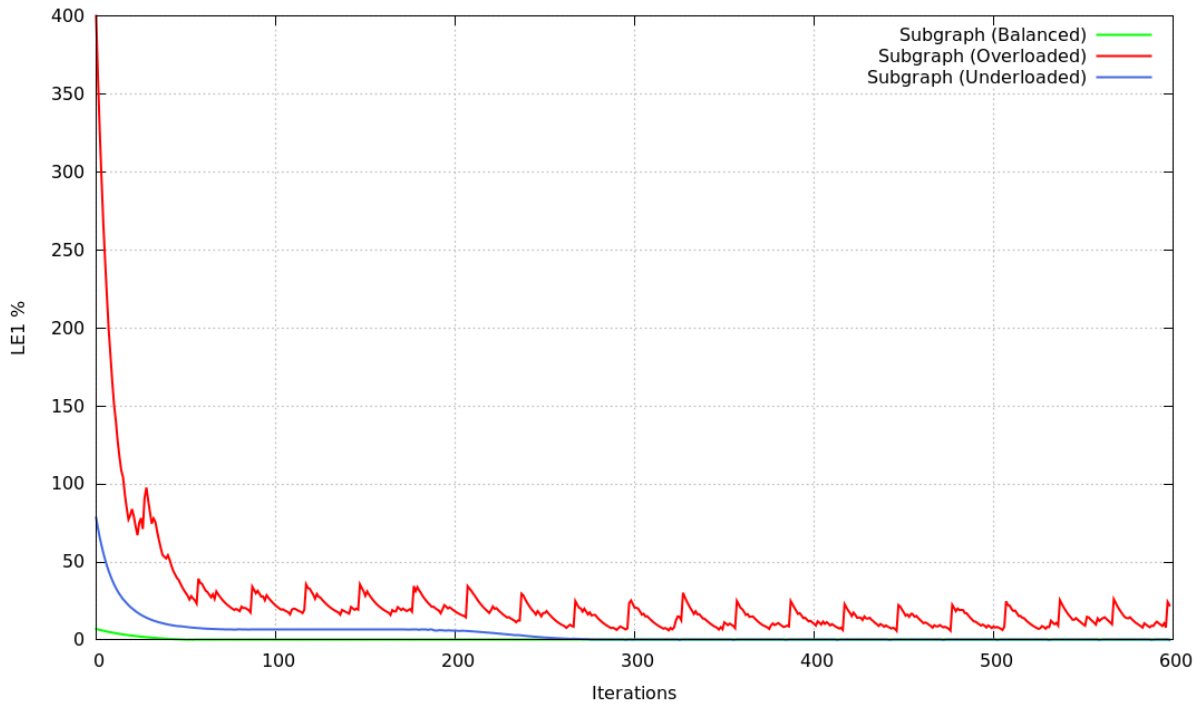


FIGURE 6.6: Average $LE1$ for subgraph scenario of the DDLB algorithm execution by considering load flexibility ($LF = 1$) or not ($LF = 0$) and considering the voltage flexibility ($VF = 1$) or not ($VF = 0$).

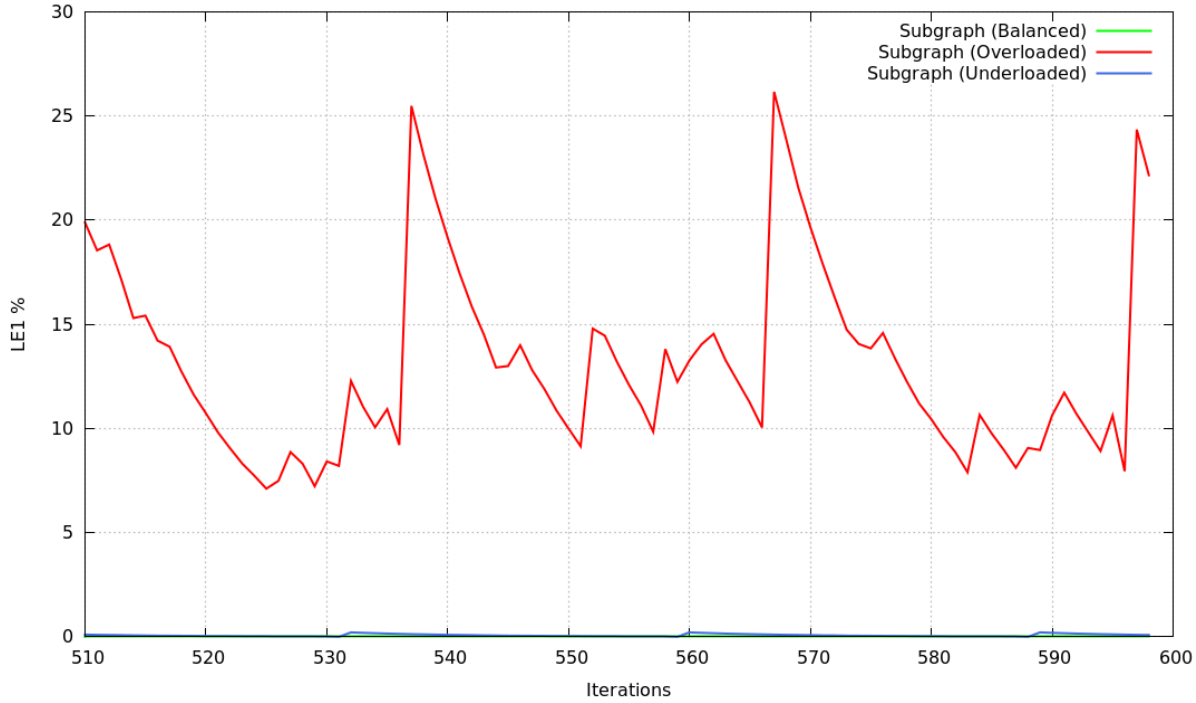


FIGURE 6.7: Average $LE1$ for subgraph scenario of the DDLB algorithm execution for iteration 510–600

TABLE 6.6: Flexibility variations for balanced inputs

Average of measure (%)	(LF,VF)=(0,0)	(LF,VF)=(0,1)	(LF,VF)=(1,0)	(LF,VF)=(1,1)	One-Shot
VE1	0.00	0.09	0.00	0.12	45.95
VE2	0.39	0.46	0.40	0.47	7.89
LE1	0.00	0.00	0.02	0.02	0.00
LE2	14.83	15.00	24.77	22.08	0.00
SE	185.88	186.64	185.88	186.64	193.27

TABLE 6.7: Flexibility variations for overloaded inputs

Average of measure (%)	(LF,VF)=(0,0)	(LF,VF)=(0,1)	(LF,VF)=(1,0)	(LF,VF)=(1,1)	One-Shot
VE1	1.43	1.72	1.09	1.47	58.44
VE2	0.50	0.55	0.48	0.55	14.42
LE1	7.83	8.00	17.77	15.08	0.00
LE2	14.76	15.21	24.27	21.59	0.00
SE	161.37	161.95	161.25	161.83	138.86

For the balanced case (Table 6.6), there is really no significant difference between the flexibility variation, except for $LE2$. The measure $LE2$ does not reward load flexibility, so a larger $LE2$ for $LF = 1$ is understandable. Similarly in the underloaded case Table 6.8 the differences are insignificant. In the overloaded case, however, there are more differences in $LE1$ and $VE1$ across flexibility variations.

Further study may be needed to see if the DDLB algorithm can automatically transition from one scenario to another, accounting for flexibility variations across these transitions.

Figures 6.8 through 6.19 show the different flexibility variation based convergence comparison graphs applied to balanced, overloaded and underloaded scenarios. Note that "SUB" on the legend of each graph points to Subgraph. SUB=0 indicates subgraph not used, SUB=1 indicates Subgraph used.

TABLE 6.8: Flexibility variations for underloaded inputs

Average of measure (%)	(LF,VF)=(0,0)	(LF,VF)=(0,1)	(LF,VF)=(1,0)	(LF,VF)=(1,1)	One-Shot
VE1	0.00	0.19	0.03	0.27	58.43
VE2	0.39	0.46	0.40	0.47	6.77
LE1	0.00	0.00	0.36	0.41	0.00
LE2	0.00	0.00	0.38	0.42	0.00
SE	156.43	156.04	156.34	155.95	154.87

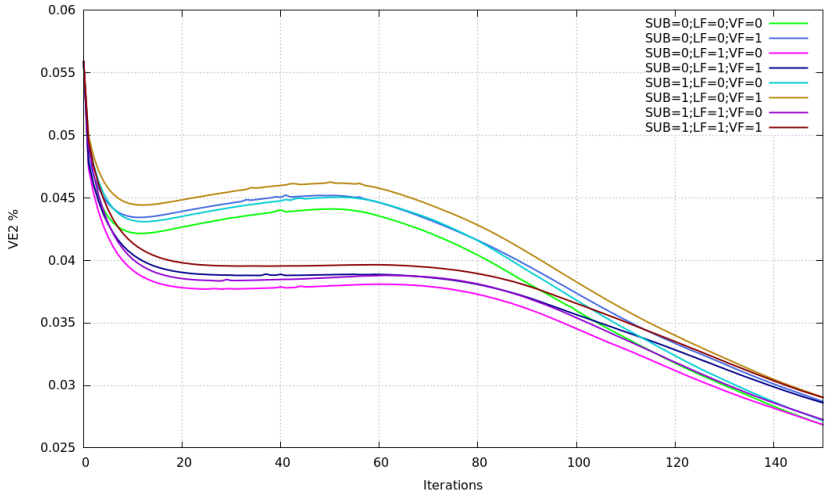


FIGURE 6.8: VE2 for balanced inputs for iteration 0–150

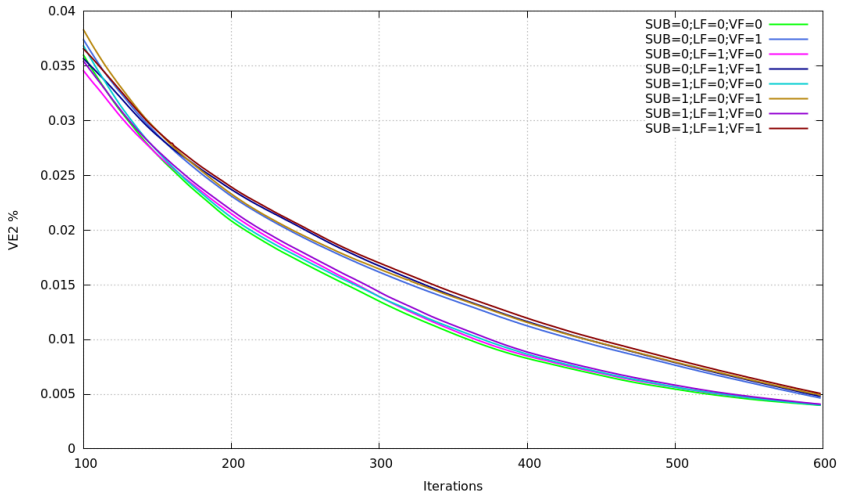


FIGURE 6.9: VE2 for balanced inputs for iteration 100–600

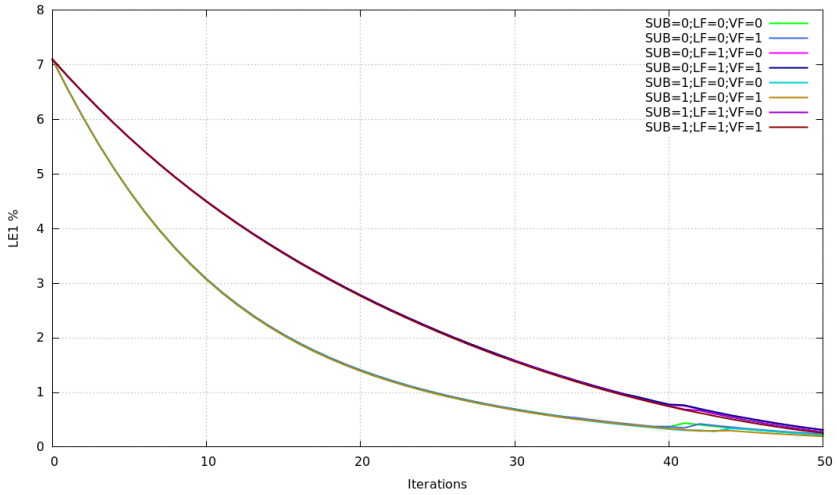


FIGURE 6.10: LE1 for balanced inputs for iteration 0–150

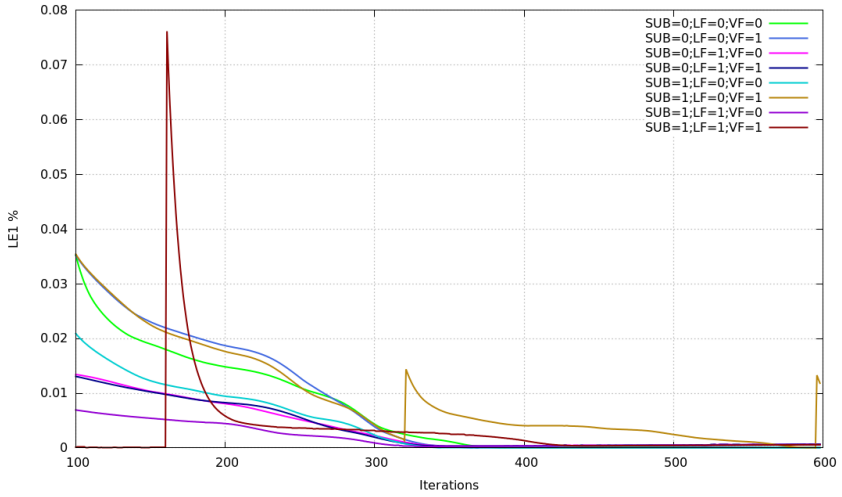


FIGURE 6.11: LE1 for balanced inputs for iteration 100–600

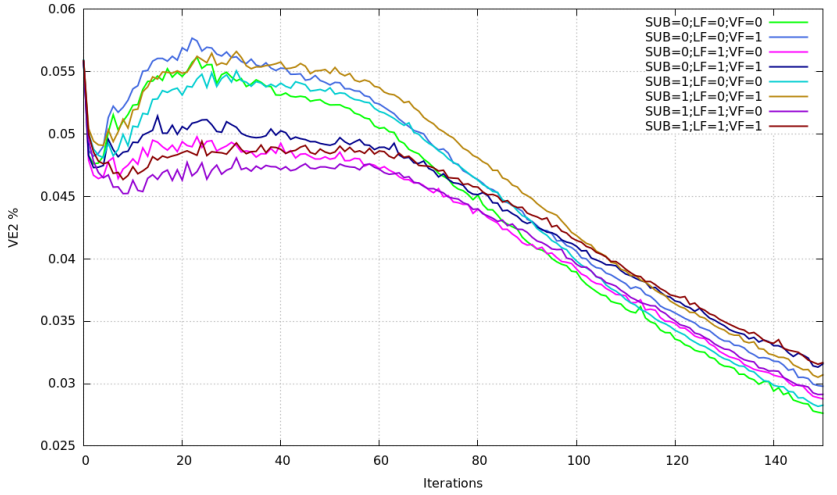


FIGURE 6.12: VE2 for overloaded inputs for iteration 0–150

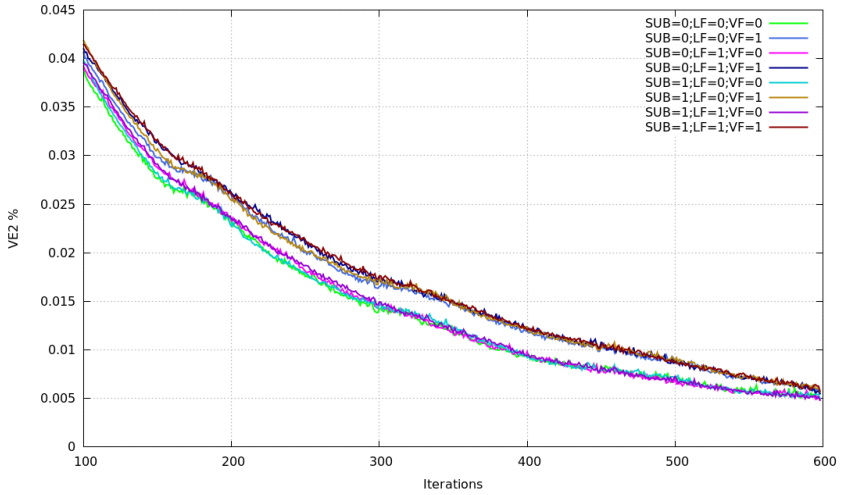


FIGURE 6.13: VE2 for overloaded inputs for iteration 100–600

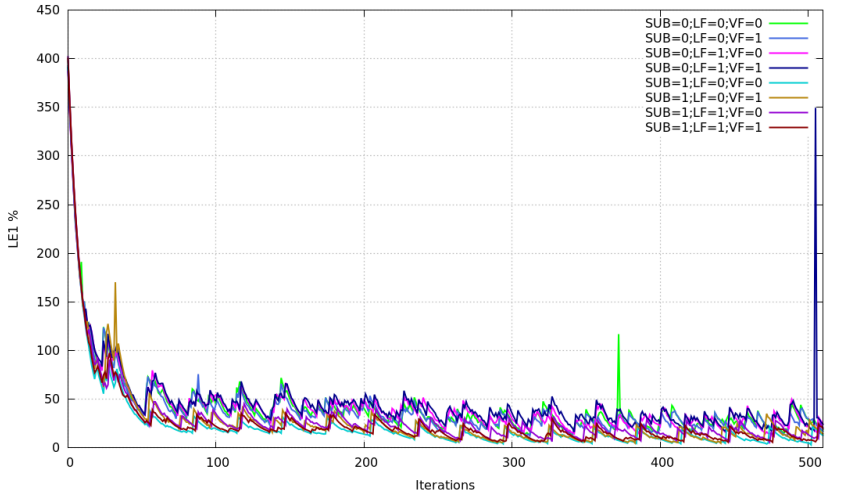


FIGURE 6.14: LE1 for overloaded inputs for iteration 0–510

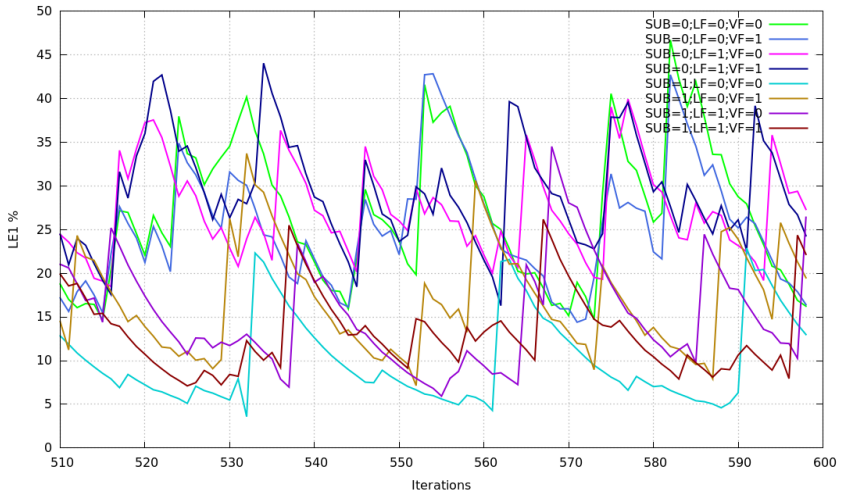


FIGURE 6.15: LE1 for overloaded inputs for iteration 510–600

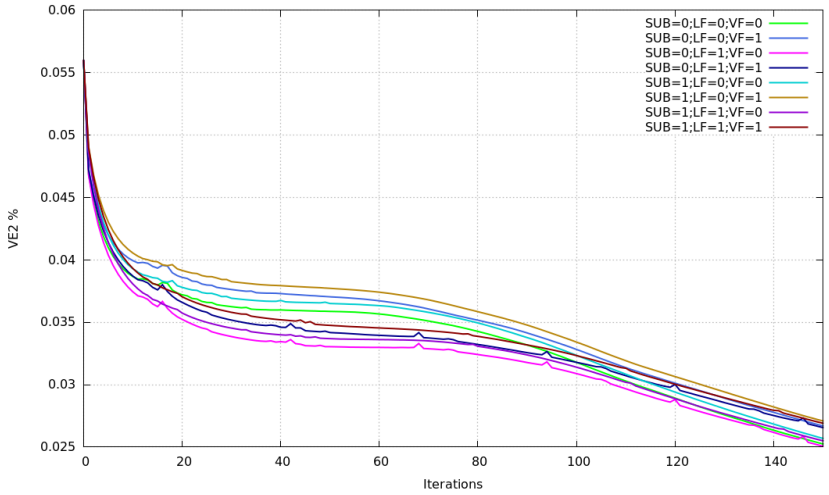


FIGURE 6.16: VE2 for underloaded inputs for iteration 0–150

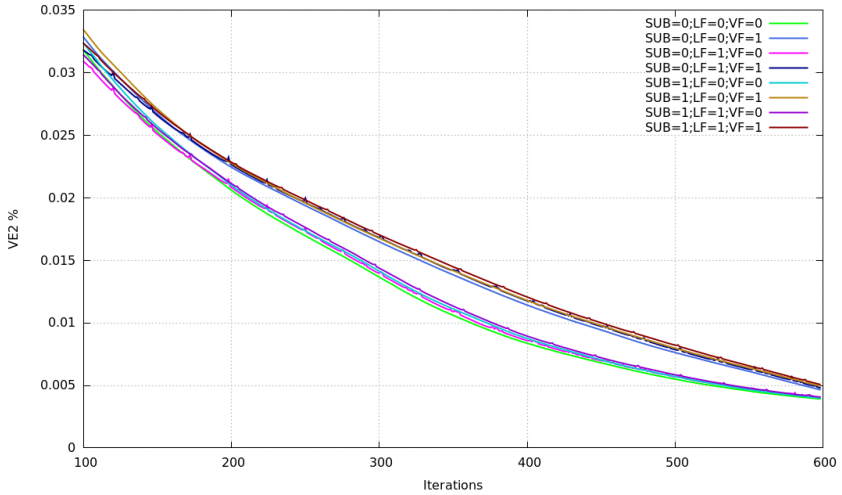


FIGURE 6.17: VE2 for underloaded inputs for iteration 100–600

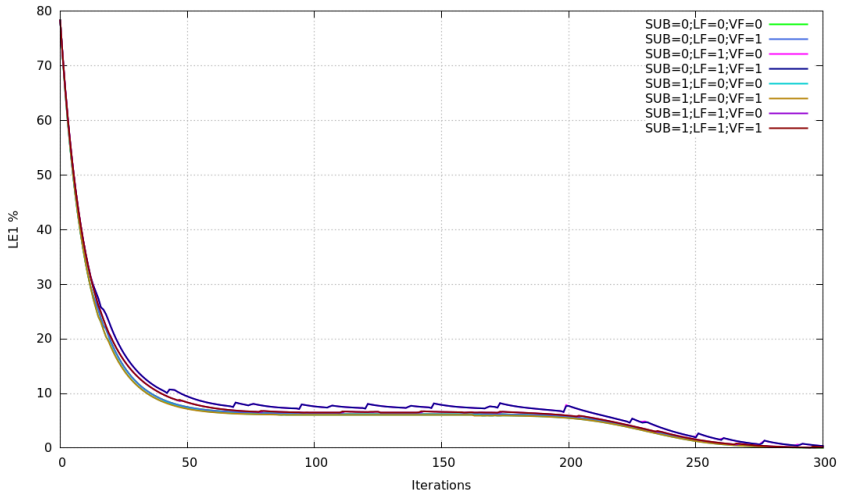


FIGURE 6.18: LE1 for underloaded inputs for iteration 0–300

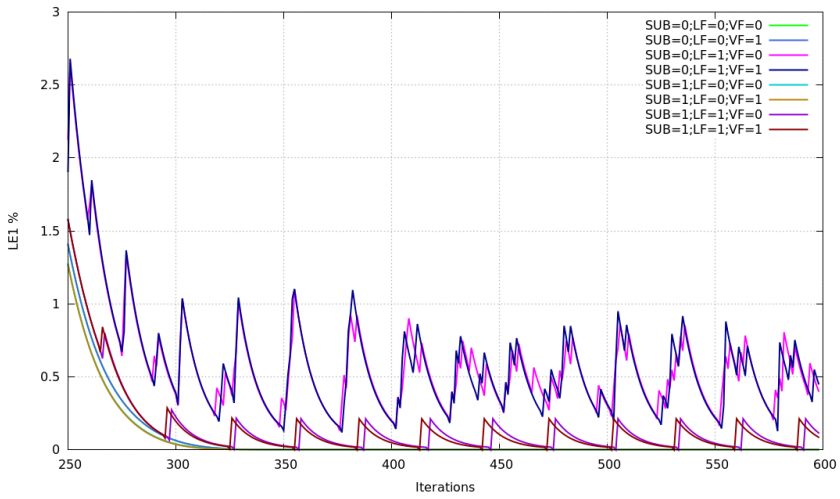


FIGURE 6.19: LE1 for underloaded inputs for iteration 250–600

7

Conclusions

In this thesis we proposed a distributed algorithm based on diffusion to balance loads on an electrical power grid, while maintaining stable operation. This algorithm is called the Diffusion-driven Distributed Load Balancing (DDLB) algorithm. We implemented the DDLB algorithm on the OMNET++ Discrete Event Simulator. The response of the physical grid was simulated on a load flow program. Together, the OMNET++ and load flow programs simulated a deployment of the DDLB algorithm on the grid.

Through extensive simulations of around 1.6 million input instances we showed the superiority of the DDLB algorithm over a "one-shot" algorithm that balances loads without regard for the system stability. We show that the DDLB algorithm was stable for over 99% of the inputs tested. For the same set of inputs, the one-shot algorithm produced a stable solution less than 43% of the time.

Our simulations included a study of several scenarios that a grid could be subjected to. These include balanced, overloaded, underloaded, radially damaged and communication subgraphs. The last two scenarios represent cases where power sources were disabled around a point, and one in which the communication network was substantially compromised. In all these scenarios the DDLB algorithm outperformed the one-shot algorithm.

7.1 Future work

Several directions for future work arise from our study.

The first and foremost is tuning the DDLB algorithm for better performance. In particular, the flexibility variations of Section 6.6 should be integrated into a single algorithm that automatically adjusts to the scenario at hand.

We have assumed the grid (load flow program) to respond fast (in time comparable to one iteration of the DDLB algorithm). Thus the interaction between the compute and grid planes is as shown in Figure 3.4 (see page 22). This is somewhat optimistic. In reality, there will be a delay between the application of the the prescription from the DDLB algorithm to the power grid (in our case the load flow program) and the feedback received, and a reconciliation is needed to correct the error due to the delayed feedback. This is illustrated in Figure 7.1. At time t, the compute plane will determine loads to be applied at time $t + 1$. The feedback of the application at time t is not received until all the iterations are completed for time $t + 1$. Once the feedback is received, the load values calculated are adjusted (reconciled), and then the $(t + 1)$th values are applied.

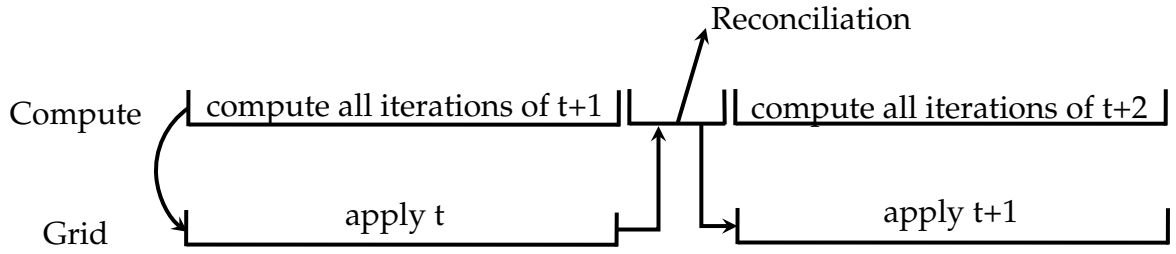


FIGURE 7.1: Actual time simulation

In a more interesting extension of the work, could the DDLB algorithm be converted to a node-level neural network through deep-learning? Could its performance be bounded by diffusion algorithms, for which analytical predictions on convergence is possible?

References

[1] C. P. Adolphs and P. Berenbrink. Improved bounds for discrete diffusive load balancing. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 820–826. IEEE, 2012.

[2] M. Ammad-uddin. Performance analysis of centralized, distributed and hybrid demand load control architecture for smart power grid. In *Energytech, 2013 IEEE*, pages 1–6. IEEE, 2013.

[3] M. Arora, S. K. Das, and R. Biswas. A de-centralized scheduling and load balancing algorithm for heterogeneous grid environments. In *Parallel Processing Workshops, 2002. Proceedings. International Conference on*, pages 499–505. IEEE, 2002.

[4] P. Berenbrink, T. Friedetzky, and Z. Hu. A new analytical method for parallel, diffusion-type load balancing. *Journal of Parallel and Distributed Computing*, 69(1): 54–61, 2009.

[5] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation computer systems*, 25(6):599–616, 2009.

[6] J. Cao, D. P. Spooner, S. A. Jarvis, S. Saini, and G. R. Nudd. Agent-based grid load balancing using performance-driven task scheduling. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 10–pp. IEEE, 2003.

[7] J. Cao, D. P. Spooner, S. A. Jarvis, and G. R. Nudd. Grid load balancing using intelligent agents. *Future generation computer systems*, 21(1):135–149, 2005.

[8] S. Caron and G. Kesidis. Incentive-based energy consumption scheduling algorithms for the smart grid. In *Smart grid communications (SmartGridComm), 2010 First IEEE international conference on*, pages 391–396. IEEE, 2010.

[9] Y.-C. Chang, T.-Y. Lee, C.-L. Chen, and R.-M. Jan. Optimal power flow of a wind-thermal generation system. *International Journal of Electrical Power & Energy Systems*, 55:312–320, 2014.

[10] R. Chedid, H. Akiki, and S. Rahman. A decision support technique for the design of hybrid solar-wind power systems. *Energy conversion, ieee transactions on*, 13(1): 76–83, 1998.

[11] J. Chen, S.-Y. Tu, and A. H. Sayed. Distributed optimization via diffusion adaptation. In *Computational Advances in Multi-Sensor Adaptive Processing (CAMSAP), 2011 4th IEEE International Workshop on*, pages 281–284. IEEE, 2011.

[12] Y. Chen and A. Luo. A robust predictive dual-loop control strategy for grid-connected photovoltaic generation system. In *Power Electronics and Motion Control Conference (IPEMC), 2012 7th International*, volume 3, pages 2007–2011, June 2012. doi: 10.1109/IPEMC.2012.6259149.

[13] Entergy. Entergy louisiana - summer savings pilot program, 2015. URL `http://www.entergy-louisiana.com/ssp`. [Online; accessed 09-April-2015].

[14] ETAP. Load flow software, 2015. URL `http://etap.com/load-flow-analysis/load-flow-analysis.htm`. [Online; accessed 23-March-2015].

[15] H. Farhangi. The path of the smart grid. *Power and Energy Magazine, IEEE*, 8(1): 18–28, 2010.

[16] C. Ibars, M. Navarro, and L. Giupponi. Distributed demand management in smart grid with a congestion game. In *Smart grid communications (SmartGridComm), 2010 first IEEE international conference on*, pages 495–500. IEEE, 2010.

[17] S. Kar and J. Moura. Consensus + innovations distributed inference over networks: cooperation and sensing in networked systems. *Signal Processing Magazine, IEEE*, 30(3):99–109, May 2013.

[18] H. Kazari, A. Abbaspour-Tehrani Fard, A. Dobakhshari, and A. Ranjbar. Voltage stability improvement through centralized reactive power management on the smart grid. In *IEEE PES Innovative Smart Grid Technologies (ISGT)*, 2012.

[19] A. Kiani and A. Annaswamy. Distributed hierarchical control for renewable energy integration in a smart grid. In *Innovative Smart Grid Technologies (ISGT), 2012 IEEE PES*, pages 1–8. IEEE, 2012.

[20] S. Kumar, A. Jantsch, J.-P. Soininen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja, and A. Hemani. A network on chip architecture and design methodology. In *VLSI, 2002. Proceedings. IEEE Computer Society Annual Symposium on*, pages 105–112. IEEE, 2002.

[21] S.-Y. Lin and J.-F. Chen. Distributed optimal power flow for smart grid transmission system with renewable energy sources. *Energy*, 56(0):184 – 192, 2013.

[22] A. Mohsenian-Rad, V. W. Wong, J. Jatskevich, and R. Schober. Optimal and autonomous incentive-based energy consumption scheduling algorithm for smart grid. In *Innovative Smart Grid Technologies (ISGT), 2010*, pages 1–6. IEEE, 2010.

[23] A. Mohsenian-Rad, V. W. Wong, J. Jatskevich, R. Schober, and A. Leon-Garcia. Autonomous demand-side management based on game-theoretic energy consumption scheduling for the future smart grid. *Smart Grid, IEEE Transactions on*, 1(3): 320–331, 2010.

[24] M. Palmer, M. Tachibana, T. Senjyu, T. Funabashi, A. Y. Saber, and M. Datta. Grid stabilization with decentralized controllable loads using fuzzy control and droop characteristics. *International Journal of Emerging Electric Power Systems*, 15(4):357 – 365, 2014.

[25] J. Pezoa, L. García-Santander, and A. Cárcamo-Gallardo. A scalable distributed algorithm for optimizing distribution power-grids. In *Chilean Computer Science Society (SCCC), 2009 International Conference of the*, pages 18–24, Nov 2009.

[26] P. Ram and A. Jha. Automatic generation control of interconnected hydro-thermal system in deregulated environment considering generation rate constraints. In *Industrial Electronics, Control Robotics (IECR), 2010 International Conference on*, pages 148–159, Dec 2010.

[27] P. Samadi, A.-H. Mohsenian-Rad, R. Schober, V. W. Wong, and J. Jatskevich. Optimal real-time pricing algorithm based on utility maximization for smart grid. In *Smart Grid Communications (SmartGridComm), 2010 First IEEE International Conference on*, pages 415–420. IEEE, 2010.

[28] M. Shinwari, A. Youssef, and W. Hamouda. A water-filling based scheduling algorithm for the smart grid. *Smart Grid, IEEE Transactions on*, 3(2):710–719, 2012.

[29] A. G. Tsikalakis and N. D. Hatziargyriou. Centralized control for optimizing microgrids operation. In *Power and Energy Society General Meeting, 2011 IEEE*, pages 1–8. IEEE, 2011.

[30] A. Varga. The omnet++ discrete event simulation system. In *In ESM'01*, 2001.

[31] F. F. Wu, K. Moslehi, and A. Bose. Power system control centers: Past, present, and future. *Proceedings of the IEEE*, 93(11):1890–1908, 2005.

[32] H. Zeineldin, E. El-Saadany, and M. Salama. Distributed generation micro-grid operation: control and protection. In *Power Systems Conference: Advanced Metering, Protection, Control, Communication, and Distributed Resources, 2006. PS'06*, pages 105–111. IEEE, 2006.

[33] N. A. Ziv and E. G. Tiedemann Jr. Method and apparatus for providing centralized power control administration for a set of base stations, Mar. 16 1999. US Patent 5,884,187.

Appendix

The code for the DDLB load balancing algorithm is provided in this Appendix.

filename : distAlg.cc

```
#include "distAlg.h"

Define_Module(DistAlg)

void DistAlg::activity()
{
   if(exec_once){ //run only once for entire network
      doDiffCommunication = int(par("dodiffCommunication"));
      zeroloadslack      =int(par("zeroloadslack"));
      zerovoltslack      =int(par("zerovoltslack"));

      if(getLinevals(par("line_filename"))){
         callFinish();
         endSimulation();
      }

      if(getBusvals(par("bus_filename"))){
         callFinish();
         endSimulation();
      }

      for(int i =0;i < NUM;++i){
         loadCost[i]=0;
      }

      string c = par("bus_filename");
      statsStream << basename((char*)c.c_str()) << ",";

      singleRunLoadFlow(); // run once
      exec_once = false;
   }

   _ID = getIndex();
   _heardFromNeighbors=0;
   _localLoadCost=0;
   _vCost=0;
   _lCost=0;
   _degree = gateSize("gate$o");
   powers[_ID]= stable_powers[_ID];
```

```
uint flag=1;
bool init=true,
      comm_active=true;

double sumLocalLoadcosts = 0,
      sumLocalVoltcosts = 0;

//initialization
while(init){

   ++initcalls;
   //doDiffCommunication = par("DIFF_COMM");
   doDiffCommunication = int(par("dodiffCommunication"));

   rpowers[_ID]=powers[_ID]*0.2; // 20% of active power set as
      reactive power initially

   calculateLocalLoadcost(_lCost,asked_power[_ID],powers[_ID]);

   if(initcalls==NUM){
      callMatlab();
      ++rounds;
   }

   for(uint k=_degree;k--;)
      send(DistAlg::genMsg(powers[_ID]), "gate$o", k);

   init=false;
}

while(comm_active){
   msg = check_and_cast<valmsg *>(receive());

   powers[_ID] = mat_powers_c[_ID];
   calculateLocalLoadcost(_lCost,asked_power[_ID],powers[_ID]);

   if(flag==0){
      flag=1;
      for(uint k=_degree;k--;)
         send(genMsg(powers[_ID]), "gate$o", k);
   }

   if( flag ) {  //hear from neighbours

      if(doDiffCommunication){
         if(rounds==1){ //coin toss for diff comm topology edge
            drops
            if(rand()%9 < 5){      //33% drop rate
               sumLocalVoltcosts += Vscale(msg->getSrcVolt());
```

```
            sumLocalLoadcosts += (msg->getSrc()==slack_bus_ID)
                              ? 0 : msg->getSrcLCost();
         }else{
            nbrToBeDroppedList.push_back(msg->getSrc());
         }
      }else{
         if( find (nbrToBeDroppedList.begin(),
            nbrToBeDroppedList.end(),
                  msg->getSrc())
               != nbrToBeDroppedList.end())
         {}else{
            sumLocalVoltcosts += Vscale(msg->getSrcVolt());
            sumLocalLoadcosts += (msg->getSrc()==slack_bus_ID)
                              ? 0 : msg->getSrcLCost();
         }
      }
   }else{
      sumLocalVoltcosts += Vscale(msg->getSrcVolt());
      sumLocalLoadcosts += (msg->getSrc()==slack_bus_ID)
                        ? 0 : msg->getSrcLCost();
   }
   ++_heardFromNeighbors;

   if(_heardFromNeighbors == _degree) {// heard from all nbrs
      _heardFromNeighbors=0;

      calculateLoadcost(loadCost[_ID],sumLocalLoadcosts
                  ,_lCost);

      calculateVoltagecost(_vCost,sumLocalVoltcosts);

      calculateNewLoad(powers[_ID],rpowers[_ID],loadCost[_ID]
                   ,_vCost);

      resets(flag,sumLocalLoadcosts,sumLocalVoltcosts);

      ++nodesDone;
   }

   if(nodesDone == NUM) { // all nodes compute done
      callMatlab();
      ++ rounds ;

      if(rounds == RNDS) {
         writetoFileandFinish();
      }
      flag=1;
      calculateLocalLoadcost(_lCost,asked_power[_ID],
                        powers[_ID]);

      for(uint k=_degree;k--;) //send to neighbours
```

```
                send(genMsg(mat_powers_c[_ID]), "gate$o", k);

             nodesDone = 0;
          }
       }
       delete msg;
    }
}

valmsg* DistAlg::genMsg(double &Srcval){

    valmsg *msg = new valmsg("");
    msg->setSrc(_ID);
    msg->setSrcValue(Srcval);
    msg->setSrcVolt(real_volts[_ID]);
    msg->setSrcLCost(_lCost);
    return msg;
}

void DistAlg::calculateNewLoad(double &load, double &rpower,
                        const double &loadCost, const double
                           &voltCost){


    double _loadScale,_voltScale,_closeToZero,
          _maxLoadChangeFactor;
    _loadScale        = 50;
    _voltScale        = 30;
    _maxLoadChangeFactor = 0.1;
    _closeToZero      = 0.0000001 ;

    if(loadCost < 0) {
       if(load>= -_closeToZero && load <=_closeToZero)
          load = _closeToZero*10;
       load += ( fabs(load) *
          min(_maxLoadChangeFactor,(fabs(loadCost)/_loadScale))) ;
    }

    if(loadCost >= 0) {
       if(load>= -_closeToZero && load <=_closeToZero)
          load = -_closeToZero*10;
       load -= (fabs(load) *
          min(_maxLoadChangeFactor,(fabs(loadCost)/_loadScale)) );
    }

    if(voltCost > 0){
       if(rpower>= -_closeToZero && rpower <=_closeToZero)
          rpower = _closeToZero*10;
       rpower += fabs(rpower)*(fabs(voltCost)/_voltScale) ;
    }
```

```
    if(voltCost <= 0){
        if(rpower>= -_closeToZero && rpower <=_closeToZero)
            rpower = -_closeToZero*10;
        rpower -= fabs(rpower)*(fabs(voltCost)/_voltScale) ;
    }

}

void DistAlg::calculateLocalLoadcost(double &l_cost, const double
    &tval, const double &load){

    l_cost = Lscale(load,tval) ;
}

void DistAlg::calculateVoltagecost(double &v_cost, const double
    &sum_local_voltcosts){

    if(std::find(gens.begin(), gens.end(), _ID ) != gens.end())
        v_cost = 0;
    else
        v_cost = degreeBasedAveraging(sum_local_voltcosts,
                                Vscale(real_volts[_ID]));
}

void DistAlg::calculateLoadcost(double &loadCost, const double
    &sum_local_loadcosts, const double &l_lcost){

    loadCost = degreeBasedAveraging(sum_local_loadcosts,l_lcost);
}

void DistAlg::endlOpenWriteCloseFile(string fileName,
    std::stringstream& Stream){
    Stream << endl;
    ofstream myfile(fileName, ios::out | ios::app);
    myfile << Stream.rdbuf();
    myfile.close();
}

void DistAlg::writetoFileandFinish(){

    //write to file
    endlOpenWriteCloseFile("Stats_collection.csv",statsStream);
    endlOpenWriteCloseFile("maxVoltError.csv",maxVoltStream);
    endlOpenWriteCloseFile("avgVoltError.csv",avgVoltStream);
    endlOpenWriteCloseFile("maxPerfLoadError.csv",maxPerfLoadStream);
    endlOpenWriteCloseFile("avgPerfLoadError.csv",avgPerfLoadStream);
    endlOpenWriteCloseFile("maxLoadVoltError.csv",maxLoadVoltStream);

    callFinish();
    endSimulation();
}
```

```
double DistAlg::degreeBasedAveraging(const double &sumOfOthers,
   const double &ownValue){
   double a = 27/2,
         b = 3/2,
         f_degree = 1 + a/pow(b,_degree);

   return (sumOfOthers + ( f_degree*ownValue ))
         /(_degree + f_degree);
}

double DistAlg::Vscale(const double &val){ // local voltage cost
   actually

   double a,b,c,Delta,Delta1,C,del,del1,del2,del3,del4;
   C=2; //scaling from slope 2 to 3
   del=0.01;
   del1 = del2 = del3 = del4 =del;
   c=max_volt[_ID];
   a=min_volt[_ID];
   b=bus[NUM + _ID]; //ideal voltages

   Delta = dynamicDelta(0.5,1.2);
   Delta1 = dynamicDelta(0.25,0.6);

   return
      GenericScale(a,b,c,Delta,Delta1,C,del1,del2,del3,del4,val);
}

double DistAlg::Lscale(const double&val,const double&tval){ //
   local LOAD cost
   double a,b,c,Delta,Delta1,C,alpha,del1,del2,del3,del4;
   C=2;
   alpha = 0.10;
   c=max_power[_ID];
   a=min_power[_ID];

   if(a==0)
      del1 = del2 = fabs(c-a)/10;
   else
      del1 = del2 = fabs(alpha*a);
   if(c==0)
      del3 = del4 = fabs(c-a)/10;
   else
      del3 = del4 = fabs(alpha*c);

   b=tval;    //ideal voltages

   Delta = dynamicDelta(0.05,0.2);
   Delta1 = dynamicDelta(0.2,0.6);
```

```
    return
        GenericScale(a,b,c,Delta,Delta1,C,del1,del2,del3,del4,val);
}

double DistAlg::GenericScale(const double& a,const double&b,const
    double&c,
                        const double&Delta,const double&Delta1,
                        const double&C,
                        double &del1,double &del2,double &del3,double
                            &del4,
                        const double&val
                        ) {

    double
        out,a1,c1,s1,ya,yc,s2,gammab,gammac,gammad,gammae,s3,s4,s5;

    a1=a - del2;
    c1=c + del3;

    if(c==b && b==a)
        s1 =0;
    else
        s1=fabs(Delta/max(fabs(c-b),fabs(b-a))); //line b
    ya=s1*(b-a);
    yc=s1*(c-b);

    if(del1 == 0)
        del1 = 0.1;
    s2 = (Delta1 / del1); //line a
    gammab = -ya;

    s3 = (C*s2) ; //line c
    gammac = -ya - Delta1;

    if(del3 == 0)
        del3 = 0.1;
    s4 = (Delta1 / del3);
    gammad = yc ;

    s5 = (C*s4);
    gammae = yc + Delta1 ;

    if(val < a1)
        out=s3*((val)-a1) + (gammac);
    else if(val >= a1 && val < a)
        out=s2*((val)-a) + (gammab);
    else if(val >= a && val < c)
        out =s1*((val)-b);
    else if(val >= c && val < c1)
        out=s4*((val)-c) + (gammad);
    else // > c1
```

```
        out=s5*((val)-c1) + (gammae);

    return out;
}
void DistAlg::resets(uint&flag, double&sum_local_loadcosts,
    double&sumvoltcosts){

    flag=0;
    sum_local_loadcosts = 0;
    sumvoltcosts = 0 ;
}

double DistAlg::dynamicDelta(double first, double last){
    return (last-first)*(matlabCallCount+1)/RNDS + first;
}


void DistAlg::callMatlab() {
    loadFlowSolution();
}
```

filename : distAlg.h

```
#include <cstdlib>
#include <cstdio>
#include <cstring>
#include <omnetpp.h>
#include <cmath>
#include <iostream>
#include <iomanip>
#include <fstream>
#include "valmsg_m.h"
#include <vector>
#include <tuple>
#include <algorithm>
#include <map>
#include <libgen.h>
#include <helperfunctions.h>

extern "C" {
    #include <matlabrun.h>
}

#define STACKSIZE 262144

typedef std::vector<int> vectorInt;
const int NUM   = 30;
const int lineNUM = 40;
const int SETWIDTH = 14;
const int RNDS  = 600;
const vectorInt printRounds = {1,150,300,450,600};
using namespace std;
```

```
class DistAlg : public cSimpleModule
{
 private:
   enum BusType {slack=1,gen,load};

   valmsg *msg;
   vectorInt nbrToBeDroppedList;
   bool doDiffCommunication ,
     zeroloadslack,
      zerovoltslack;
   uint _ID,
      _heardFromNeighbors,
      _degree;
   double _localLoadCost,
        _vCost,
        _lCost;

 public:
   DistAlg() : cSimpleModule(STACKSIZE) {}
   ~DistAlg();
   virtual void activity();
   virtual valmsg* genMsg(double &Srcval);
   void calculateNewLoad(double &load,double &rpower,const double
      &loadCost,const double &voltCost);
   void calculateLocalLoadcost(double &l_cost,const double
      &tval,const double &load);
   void calculateVoltagecost(double &v_cost,const double &sumvolts);
   void calculateLoadcost(double &loadCost,const double
      &sum_local_loadcosts,const double &l_lcost);
   void writetoFileandFinish();
   void
      resets(uint&flag,double&sum_local_loadcosts,double&sumvolts);
   double degreeBasedAveraging(const double &sumOfOthers, const
      double &ownValue);
   double Vscale(const double&val);
   double Lscale(const double&val, const double&tval);
   double GenericScale(const double& a,const double&b,const
      double&c,const double&Delta,
                  const double&Delta1,const double&C,double &del1,
                  double &del2,double &del3,double &del4,const
                     double&val
                  );
   void endlOpenWriteCloseFile(string fileName, std::stringstream
      &Stream);

   double dynamicDelta(double first, double last);
   void callMatlab();
   int findnoofGensAndSlack(int len, BusType type);
   int getBusvals(const std::string &input);
   int getLinevals(const std::string &input);
```

```
   void initializeBusParameters(double *bus, double *powers, double
      *rpowers,
                          uint slack_bus_ID, double &genSum, double
                             &loadSum);
   void maxAndAvg(double &max, double value, double &avg);
   void loadFlowSolution();
   void singleRunLoadFlow();
};


DistAlg::~DistAlg()
{}

double single_run_slack;
bool exec_once = true;
double bus[NUM*10];
double  max_volt[NUM],min_volt[NUM],
      max_volt_meas[NUM],min_volt_meas[NUM],
      max_power_meas[NUM],max_power[NUM],
      min_power_meas[NUM],min_power[NUM],
      asked_power[NUM],stable_powers[NUM],
     linefull[lineNUM*7];

double M,L,Slack_error,Slack_error_percentage;
double Va,Vm,V1,V2;
std::vector<int> gens;
std::vector<int>::iterator it;
uint no_Of_Gens=0; //includes slack bus
uint no_Of_slack=0;
int slack_bus_ID;
uint iter=0;

double maxPerfLoadError_Buffer[RNDS];
double loadAt600Iter =0 ,
     loadAtPresent=0;

double bus_sol[NUM*10];
bool _getBusValsDone=false;
bool _getLineValsDone=false;
uint matlabCallCount = 0,
    nodesDone = 0,
    rounds = 0,
    initcalls = 0;

string st[]={"Voltage |","Voltage Cost
   |","localloadCost|","loadCost |",
         "React_Powers |","Powers |"," %PowerError"," |
            %Perf_PowerError"};

double powers[NUM], rpowers[NUM],
      mat_powers_c[NUM],real_volts[NUM],loadCost[NUM];
```

```
std::stringstream avgVoltStream;
std::stringstream maxVoltStream;
std::stringstream maxPerfLoadStream;
std::stringstream avgPerfLoadStream;
std::stringstream maxLoadVoltStream;
std::stringstream statsStream;

int DistAlg::findnoofGensAndSlack(int len, BusType type){
     for (int i = 0; i < len; ++i){

       if (bus[NUM*9 + i] == type) {
          if(type == 1){
          ++no_Of_slack;
          gens.push_back(i);
          return i;
          }else{
            ++no_Of_Gens;
          gens.push_back(i);
          }
       }
     }
     return 0;
}

int DistAlg::getBusvals(const std::string& input){
   if(!_getBusValsDone){
     std::ifstream ifs(input);
     if(!ifs.good()){
        cout << "      ################ error opening bus_values_file
           ################" << endl;
        return 1;
     }
      for(uint i=0;i<NUM;++i){
      bus[NUM*2 + i]= bus[NUM*3 + i]= bus[NUM*4 + i]= bus[NUM*7 +
          i]= bus[NUM*8 + i]= 0 ;
     }
     std::string line;
     int num=0;
      // double dummy;
     while(std::getline(ifs, line)){ // read one line from ifs
          std::istringstream iavgVoltStream(line); // access line as
             a stream

          iavgVoltStream >> bus[num] >> bus[NUM + num] >>
             max_volt[num]
                        >> min_volt[num] >> asked_power[num]
                >> max_power[num] >> min_power[num] >> bus[NUM*9 +
                   num] >> stable_powers[num];
          max_power_meas[num]=max_power[num];
```

```
            min_power_meas[num]=min_power[num];
            max_volt_meas[num]=max_volt[num];
            min_volt_meas[num]=min_volt[num];

            if(!zeroloadslack){
               max_power[num] = asked_power[num];
               min_power[num] = asked_power[num];
            }

            if(!zerovoltslack){
               max_volt[num] = bus[NUM + num];
               min_volt[num] = bus[NUM + num];
            }

          ++num;
        }
          slack_bus_ID = findnoofGensAndSlack(NUM,slack); //find slack
             ID
          findnoofGensAndSlack(NUM,gen);    //find gen ID's
          _getBusValsDone = true;
      }
      return 0;
}

int DistAlg::getLinevals(const std::string& input){
    if(!_getLineValsDone){
      std::ifstream ifs(input);
      if(!ifs.good()){
        cout << "     ################ error opening
           line_values_file ################" << endl;
        return 1;
      }
      std::string l1;
      int num=0;
      while(std::getline(ifs, l1)){ // read one line from ifs
        std::istringstream iavgVoltStream(l1); // acceavgVoltStream
           line as a stream
        iavgVoltStream >> linefull[num] >> linefull[lineNUM + num] >>
           linefull[lineNUM*2 + num]
          >> linefull[lineNUM*3 + num] >> linefull[lineNUM*4 + num]
          >> linefull[lineNUM*5 + num] >> linefull[lineNUM*6 + num] ;
        ++num;
      }
        _getLineValsDone = true;
    }
    return 0;
}

void DistAlg::initializeBusParameters(double *bus,double
    *powers,double *rpowers,
```

```
                        uint slack_bus_ID,double &genSum,double
                           &loadSum){
    genSum = 0;
    loadSum = 0;
       for(uint i=0;i<NUM;i++){
          if (bus[NUM*9 + i] == 3.0){
             bus[NUM*5 + i] =powers[i];
             bus[NUM*6 + i] =rpowers[i];
             bus[NUM*3 + i] = bus[NUM*4+i] =0;
          }else{
             bus[NUM*3 + i] =-powers[i];
             bus[NUM*4 + i] =-rpowers[i];
             bus[NUM*5 + i] = bus[NUM*6+i] =0;
          }

          if(bus[NUM*5+i]-bus[NUM*3+i] < 0)
             loadSum+=bus[NUM*5+i]-bus[NUM*3+i];
          else{
             if(i != slack_bus_ID)
                genSum +=bus[NUM*5+i]-bus[NUM*3+i];
          }
       }
}

void DistAlg::maxAndAvg(double &max,double value,double& avg){
    if(fabs(max) < fabs(value))
       max = value;
    avg += fabs(value);
}

void DistAlg::loadFlowSolution()
{
    initializeBusParameters(bus,powers,rpowers,slack_bus_ID,M,L);

    matlabrun(bus,linefull,no_Of_Gens,NUM,slack_bus_ID+1,
            lineNUM,no_Of_slack,bus_sol);
    ++matlabCallCount;

    double maxVoltError=0,
          avgVoltError=0,
          myVoltError =0 ,
          loadError=0,
          perfLoadError=0,
          maxLoadError=0,
          avgLoadError=0,
          maxPerfLoadError=0,
          avgPerfLoadError=0,
          maxLoadVoltError=0,
          loadVoltError=0,
          avgLoadVoltError=0,
          maxVolt=0,
```

```
      avgVolt=0,
      voltCorr=0,
      reactMax=0,
      reactAvg=0,
      rpower=0,
      pass;


for(int i=0;i<NUM;i++){
   mat_powers_c[i] = bus_sol[NUM*5 + i] - bus_sol[NUM*3 + i];
  // rpower = bus_sol[NUM*6 + i] - bus_sol[NUM*4 + i];
   real_volts[i] = bus_sol[NUM + i];
   myVoltError = (real_volts[i]>max_volt_meas[i])?(real_volts[i]
      - max_volt_meas[i]) : (
                (real_volts[i]<min_volt_meas[i]) ?
                   (real_volts[i] - min_volt_meas[i]) : 0 );
   if(i==slack_bus_ID){
      perfLoadError=0;
      loadError=0;
   }else{
      if(asked_power[i] == 0)
         loadError= (mat_powers_c[i]-asked_power[i]) ;
      else
         loadError= (mat_powers_c[i]-asked_power[i]) /
            (asked_power[i])*100 ;
      if((max_power_meas[i] * min_power_meas[i]) ==0){
         perfLoadError=(mat_powers_c[i]>max_power_meas[i])
                     ?((mat_powers_c[i] - max_power_meas[i] )): (
                     (mat_powers_c[i]<min_power_meas[i])
                         ? (( mat_powers_c[i] - min_power_meas[i]
                            )) : 0 );
      }else{
         perfLoadError=(mat_powers_c[i]>max_power_meas[i])
                  ?((mat_powers_c[i] - max_power_meas[i]
                     )/max_power_meas[i]*100 ): (
                  (mat_powers_c[i]<min_power_meas[i])
                  ? (( mat_powers_c[i] - min_power_meas[i]
                     )/min_power_meas[i]*100) : 0 );
      }


   }

   loadVoltError = myVoltError*perfLoadError;
   voltCorr = real_volts[i] - 1;
   maxAndAvg(maxVolt,voltCorr,avgVolt);
   maxAndAvg(maxLoadVoltError,loadVoltError,avgLoadVoltError);
   maxAndAvg(maxVoltError,myVoltError,avgVoltError);
   maxAndAvg(maxLoadError,loadError,avgLoadError);
   maxAndAvg(maxPerfLoadError,perfLoadError,avgPerfLoadError);
```

```
   }

reactAvg     /= NUM;
avgVolt    /= NUM;
avgVoltError  /= NUM;
avgLoadError  /= NUM;
avgPerfLoadError/= NUM;

Va = avgVoltError;
Vm = maxVoltError;

Slack_error= fabs(mat_powers_c[slack_bus_ID]) - fabs(M-L);
Slack_error_percentage = (Slack_error)*100/max(fabs(M),fabs(L));

maxPerfLoadError_Buffer[matlabCallCount - 1] = maxPerfLoadError;

maxVoltStream << maxVoltError << ","; //save to maxvoltStream
avgVoltStream << avgVoltError << ","; //save to avgvoltStream

maxPerfLoadStream << maxPerfLoadError << ",";
avgPerfLoadStream << avgPerfLoadError << ",";

maxLoadVoltStream << maxLoadVoltError << ","; //load*volt error

if(find (printRounds.begin(), printRounds.end(),
   matlabCallCount) != printRounds.end()){
   statsStream.precision(5);
    statsStream << std::fixed
         //<< std::cout << std::setprecision(5)
         << Slack_error_percentage     << ","
         << (mat_powers_c[slack_bus_ID])   << ","
         << maxVoltError         << ","
         << avgVoltError         << ","
         << (Vm/V1)                << ","
         << (Va/V2)              << ","
         << maxVolt              << ","
         << avgVolt                << ","
         << maxLoadVoltError       << ","
         << maxLoadError         << ","
         << avgLoadError         << ","
         << maxPerfLoadError       << ","
         << avgPerfLoadError       << ","
           ;
}

if(matlabCallCount == RNDS)
{

      double reactAvg =0 ;
      for(int i=0;i<NUM;i++)
```

```
            {
               //mat_powers_c[i] = bus_sol[NUM*5 + i] - bus_sol[NUM*3
                  + i];
               rpower = bus_sol[NUM*6 + i] - bus_sol[NUM*4 + i];
               if(bus_sol[9+i] != 1)
                  reactAvg += fabs(rpower);
            }
            reactAvg /= NUM;

            for(int j=(RNDS-1);j>=0;j--){
            if((fabs(maxPerfLoadError_Buffer[j])) //5% error in value
                  > (0.95 * (fabs(maxPerfLoadError)) ) ){
               statsStream << j << ",";
               statsStream << reactAvg << ",";
               break;
            }
         }
      }
   }

void DistAlg::singleRunLoadFlow()
{
   uint _percentage;
   double overallMinimumMaxVolt=20;
   double maxVolt,optimumRpowerPercentage=20;
   double avgVolt=0;
   uint _count=0;
   _percentage = 0;
   double voltCorr;

   double avgLoad=0;
   double load=0;
   double maxVoltError=0, avgVoltError=0, voltError=0;

   do{
      maxVolt = 0 ;
      avgVolt = 0;
      for(uint i=0;i<NUM;i++){
         rpowers[i] = asked_power[i]*_percentage/100;
      }
      initializeBusParameters(bus,asked_power,rpowers,
                        slack_bus_ID,M,L);
      matlabrun(bus,linefull,no_Of_Gens,NUM,slack_bus_ID+1,
              lineNUM,no_Of_slack,bus_sol);

      avgLoad = 0;
      for(uint i=0;i<NUM;++i){
         load = bus_sol[NUM*5 + i] - bus_sol[NUM*3 + i];
         avgLoad += load;
         voltCorr = bus_sol[NUM + i] - 1;
         maxAndAvg(maxVolt,voltCorr,avgVolt);
```

```
    }

    avgLoad/=NUM;

    if(is_valid(maxVolt) && is_valid(avgLoad) &&
       (fabs(overallMinimumMaxVolt) > fabs(maxVolt) ) ){
       overallMinimumMaxVolt = maxVolt;
       optimumRpowerPercentage = _percentage;
    }

    _percentage+=10;

}while(++_count < 6);

_percentage = optimumRpowerPercentage ;

for(uint i=0;i<NUM;i++){
    rpowers[i] = asked_power[i]*_percentage/100;
}

initializeBusParameters(bus,asked_power,rpowers,
                        slack_bus_ID,M,L);
iter = matlabrun(bus,linefull,no_Of_Gens,NUM,slack_bus_ID+1,
             lineNUM,no_Of_slack,bus_sol);

single_run_slack =  bus_sol[NUM*5+slack_bus_ID] -
    bus_sol[NUM*3+slack_bus_ID];
Slack_error= fabs(single_run_slack) - fabs(M-L);
Slack_error_percentage = (Slack_error)*100/max(fabs(M),fabs(L));

maxVoltError=0, avgVoltError=0, voltError=0;
avgVolt=0;
maxVolt=0;

for(uint i=0;i<NUM;++i){
    voltError = bus_sol[NUM + i] - bus[NUM + i];
    maxAndAvg(maxVoltError,voltError,avgVoltError);
    voltCorr = bus_sol[NUM + i] - 1;
    maxAndAvg(maxVolt,voltCorr,avgVolt);
}
avgVolt/=NUM;
avgVoltError /= NUM;
V2 = avgVoltError ;
V1 = maxVoltError ;

statsStream
      << Slack_error_percentage  << ","
      << single_run_slack  << ","
      << optimumRpowerPercentage << ","
      << V1              << ","
      << V2              << ","
```

```
          << maxVolt         << ","
          << avgVolt         << ","
          << M            << ","
          << L            << ","
            ;

   maxVoltStream << V1 << ",";
   avgVoltStream << V2 << ",";
}
```

Vita

Isaac Abhilash Ayyala, a native of Bangalore, India received his Bachelor of Engineering degree at New Horizon College of Engineering, Visvesvaraya Technological University, India in 2012. He is currently a Masters student in the Division of Electrical and Computer Engineering at Louisiana State University, Baton Rouge, where he has been a graduate student since Fall 2012.