Prioritized Reconfiguration of Interdependent Critical Infrastructure Systems

David Kleppinger

Dissertation Submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
In
Computer Engineering

Robert P. Broadwater (Chairman)

Lynn Abbott

Jung-Min Park

Michael Hsiao

Manuel Perez-Quinonez

March 19, 2009
Blacksburg, Virginia

Keywords: Infrastructure Systems, Reconfiguration for Restoration, Generic Analysis, Graph Trace Analysis, Priority, Interdependence

Prioritized Reconfiguration of Interdependent Critical Infrastructure Systems

David Kleppinger

Abstract

This dissertation contains an examination of the problem of reconfiguration for restoration in critical infrastructure systems, with regard for the prioritization of those systems and the relationships between them. The complexity of the reconfiguration problem is demonstrated, and previous efforts to present solutions to the problem are discussed.

This work provides a number of methods by which reconfiguration for restoration of an arbitrary number of prioritized interdependent critical infrastructure systems can be achieved. A method of modeling systems called Graph Trace Analysis is used to enable generic operation on various system types, and a notation for writing algorithms with Graph Trace analysis models is presented.

The algorithms described are compared with each other and with prior work when run on a model of actual electrical distribution systems. They operate in a greedy fashion, attempting to restore loads in decreasing priority order. The described algorithms are also run on example models to demonstrate the ability to reconfigure interdependent infrastructure systems and systems which do not operate radially.

Acknowledgements

I would like to thank Dr. Robert Broadwater for his support and assisstance both in my research and in writing this dissertation. His unwavering belief in the value of what I was doing and the strength of my writing has been an invaluable source of confidence to me.

I would also like to thank Electric Distribution Design (EDD) Inc. for providing me with the resources necessary for me to conduct my research, in particular the Distributed Engineering Workstation (DEW) software without which I would have been unable to perform any evaluation of the validity of my work.

I would like to thank the Army, the Office of Naval Research, and the Orange and Rockland electrical utility for providing support for this research.

Furthermore I would like to thank the other members of my committee – Dr. Lynn Abbott, Dr. Jung-Min Park, Dr. Michael Hsiao and Dr. Manuel Perez-Quinonez – for their service both on my committee and as my professors in the various classes I have taken with them.

Lastly I would like to thank my friends and family for their support both as I have been working on this dissertation and throughout my life, and also for their patience with me when I was caught up with the stress of everything.

This Dissertation is dedicated

in loving memory of my Grandmother

Bessie Masson,

who was as eager to see it completed as I was.

Table of Contents

Abstract ii

Acknowledgements iii

I. Introduction 1

1.1 Objective of Reconfiguration for Restoration 1

1.2 Challenges of Interdependent System Reconfiguration 2

II. Past Work 4

2.1 A Survey of the State of the Art in Distribution System Reconfiguration for System Loss Reduction 4

2.2 Service Restoration in Naval Shipboard Power Systems 5

2.3 Restoration of Services in Interdependent Infrastructure Systems: A Network Flows Approach 7

2.4 New Approach for Distribution Feeder Reconfiguration for Loss Reduction and Service Restoration 9

2.5 A Heuristic Nonlinear Constructive Method for Distribution System Reconfiguration 10

2.6 Control Reconfiguration of Discrete Event Systems With Dynamic Control 11

2.7 Solutions Based on Genetic Algorithms and Other Evolutionary Techniques 12

2.8 Market-based Multiagent System for Reconfiguration of Shipboard Power Systems 14

2.9 Solutions based on Rules 14

2.10 Computer-based Strategy for the Restoration Problem in Electric Power Distribution 16

2.11 Optimized Restoration of Combined AC/DC Shipboard Power Systems Including Distributed Generation and Islanding Techniques 17

2.12 Contributions 18

III. Graph Trace Analysis 19
3.1 GTA Notation 19
3.2 GTA Traces 21
3.3 Component Structure for Reconfiguration and Problem Definition 23
3.4 Complexity of Prioritized Reconfiguration 27
3.5 Features of GTA 32
IV. Reconfiguration Algorithms 33
4.1 From Loads Algorithm 33
4.2 From Loads Example 35
4.3 From Sources Algorithm 37
4.4 From Sources Example 39
4.5 Hybrid Algorithm 41
4.6 Hybrid Example 42
4.7 Cotree Switch Algorithm 44
4.8 Cotree Switch Example 45
4.9 Adding Loops 46
V. Algorithm Performance 49
5.1 Example 1: Real-World Electrical Model 49
5.2 Example 2: Large Real-World Electrical Model 52
5.3 Example 3: Integrated Model 54
5.4 Example 4: Integrated Model With Loops 55
5.5 Complexity 58
5.6 Algorithm Comparison vs. Complexity Factors 65

5.7 Performance Observations 71
VI. Conclusion 73
6.1 Future Work 74
VII. References 76
Appendix A – Reconfiguration Algorithms 81

Tables and Figures

Tables

Table 1: Collection-Specific GTA Operations Used In Reconfiguration 20

Table 2: Other GTA Operations Used In Reconfiguration 21

Table 3: Seqs Created by GTA Traces 23

Table 4: Component Status Definitions 25

Table 5: Ex. 1 Algorithm Comparison in Time 50

Table 6: Ex. 1 Algorithm Comparison in Loads Serviced 50

Table 7: Ex. 1 Algorithm Comparison in Mean Phase Imbalance Across Sources 50

Table 8: Ex. 1 Algorithm Comparison in Number of Switches Operated 50

Table 9: Ex. 1 Algorithm Comparison in Mean kW Losses Across Sources and Phases 51

Table 10: Ex. 1 Algorithm Comparison in Mean kW Flow Per Source 51

Table 11: Ex. 1 Algorithm Comparison in Standard Deviation of kW Across All Sources 51

Table 12: Ex. 2 Algorithm Comparison in Time 53

Table 13: Ex. 2 Algorithm Comparison in Loads Restored 54

Table 14: Ex. 2 Algorithm Comparison in Switches Operated 54

Table 15: Complexity of Reconfiguration Functions 59

Figures

Figure 1: Simple GTA Model 22

Figure 2: AND and OR Dependencies 25

Figure 3: Constructed System *S* 28

Figure 4:Circular Interdepenency Chain 31

Figure 5: Sample GTA Model 36

Figure 6: Sample GTA Model With Failure 45

Figure 7: Real-World Electrical Model 49

Figure 8: Large Real-World Electrical Model 53

Figure 9: Second Test System – Electrical and Fluid, With Dependencies and Missions 55

Figure 10: Algorithm Results for Example 2... 56

Figure 11: Example 3 System 57

Figure 12: Example 3 Results at 100% and 75% Requirements 58

Figure 13: Ex. 3 Reconfiguration Results, 80% Dependency Requirement.. 58

Figure 14:From Loads Function Heirarchy 60

Figure 15:From Sources Function Heirarchy 61

Figure 16:Hybrid Reconfiguration Heirarchy.. 63

Figure 17: Algorithm Performance versus Number of Loads 66

Figure 18: Algorithm Performance versus Number of Switches... 67

Figure 19: Algorithm Performance versus Number of Components 68

Figure 20:Algorithm Performance versus Number of Interdependencies 68

Figure 21:Algorithm Performance versus Number of Adjacencies... 69

Figure 22:Algorithm Performance Degradation vs. Interdependencies 70

I. Introduction

Modern society depends on a set of critical infrastructure systems. These systems include electrical distribution, potable water, sewage, gas, and others. When a problem occurs in one of these systems, it can cause disruption of services not only in its system but in other systems. Often this only becomes evident during major catastrophes.

In this dissertation loads are considered to be any device which requires service from a system. Based upon the mission of the system, some loads are more important than others, and the mission of the system may change. Thus, the importance of a load may change. Reconfiguration for restoration is the process whereby disruption in services to loads is responded to and the system or systems in question are altered to restore service to the loads.

In all of these systems there are devices which can be operated to alter the system topology or even cut off entire sections of the system in order to isolate faults. These sectionalizing devices are the core of reconfiguration. In this dissertation all sectionalizing devices that are available for use, whether they be valves in fluid systems or breakers in electrical systems, will be referred to as switches.

1.1 Objective of Reconfiguration for Restoration

When faced with a reconfiguration problem, the system operator needs to be able to operate switches in order to alter service flows in the system and minimize the effects of the disruption on the system loads. System loads and the components which feed them may rely on services from other systems in order to run, and the loads themselves may differ in relative importance.

The objective of reconfiguration, therefore, is to operate system switches in order to restore loads in accordance with their importance and with respect to the various interdependencies in the system. As will be discussed in section 3.4, the problem of finding the optimal solution is NP-hard, and so the

algorithms presented in this dissertation are greedy heuristics which attempt to restore loads in decreasing priority order.

1.2 Challenges of Interdependent System Reconfiguration

Loads on a system are not of equal importance. Often there is a hierarchy of importance among system loads, and thus reconfiguration must take this prioritization into account when determining where to restore service. Furthermore the importance of a load may change based upon what the system is being asked to do, or its mission. It is typical for prior work which recognizes prioritization to limit themselves to only a few levels [1-3], but a fully robust solution must be able to handle an arbitrary number of priority levels, to the point of every load having its own unique level of importance.

Critical infrastructure systems are not independent of each other. Each has elements which depend on elements in another system. For example, water systems employ pumps. These pumps are often driven by electrical motors, which are loads on the electrical system. These intersystem dependencies must be considered by reconfiguration. Works such as Rinaldi et. al. and Lee et. al. which have looked at the question of system interdependencies (though not, in the case of Rinaldi et. al., for reconfiguration) split up dependencies into different subcategories depending on the commodity being exchanged or due to being in a geographical space [4,5]. This categorization is unnecessary for reconfiguration analysis, as any dependency between two components in different systems can be represented by the percent of service needed of one component by the other. This is the method adopted in this dissertation.

In addition to those challenges, prioritized reconfiguration is a difficult combinatorial problem. For a system with n switches, there are 2^n possible system states which could be the optimal system state. Prioritized reconfiguration is in fact NP-hard, as will be demonstrated in 3.4.

The work here considers a “system of systems” model. Critical infrastructure systems are interdependent with one another, and a disruption in one system often results in a disruption in another system. For valid solutions these interdependencies must be considered. An electrical outage can result in an outage in the potable water system, and depending upon the length of the outage, the potable water system may need to be flushed for some period of time prior to using the water. In some systems electrical power equipment depends upon cooling water. If the cooling water suffers a disruption, after some period of time the electrical equipment needs to be turned off or suffer failure due to overheating.

II. Past Work

2.1 A Survey of the State of the Art in Distribution System Reconfiguration for System Loss Reduction

-R.J. Sarfi, M.M.A. Salama, A.Y. Chikhani

The authors of this paper examined the previous work that had been accomplished in examining the reconfiguration problem. They were able to classify the solutions they found into three basic categories: methods which blended heuristics and optimization, pure heuristics, and AI-based methods.

Methods blending heuristics and optimization were heavily influenced by the algorithm developed by Merlin and Black [6], and later improved by Shirmohammadi and Hong [7]. These algorithms looked at the loops in the system and determined which switch on the loop should be turned off for the least loss of power. Loops could be examined all at once as in [7], or one at a time as in [8]. The solution proposed by Huddleston et. al. deviated from this single-switch approach by simplifying the flow model to the point where a matrix solver could be used to determine all of the required switch operations at once [9]. Another similar solution by Broadwater et. al. was cited for its improved ability to use actual system measurements [10]. Later methods developed after this survey's publication such as those proposed by [11] and [12] operate in a similar fashion, with incremental improvements mostly in determining network flows. These methods are all restricted to radial electrical systems, and only seek to minimize power losses.

Pure heuristic methods discussed in the paper seek to prune the number of possible states through a set of rules. The remaining space would be examined to determine a solution. The most prominent of these was the solution developed by Civanlar et. al. [13], which utilized a pair of rules to eliminate switches from being operable, and then used an approximation for load flows to examine the remaining switch combinations for pairs of switches which could be operated to maintain radiality and most reduce losses. Other techniques discussed in the survey presented merely incremental improvements over the

algorithm by Civanlar et. al. [13]. An heuristic developed by Lee et. al. [14] shortly after the survey was written similarly sought to operate pairs of switches to reduce losses. Rather than apply the rules proposed by Civanlar et. al. [13], the solution by Lee et. al. simply picks the most reductive pair it can find at each step, acting like a depth-first search [14]. Castro and Watanabe, rather than look at the best operation at each step like Civanlar et. al., instead operated as many viable options at a given step as were feasible in an effort to avoid local minima, but a global optimum was still not assured [15]. As with the blended heuristic/optimization algorithms, none of these algorithms look at load priority, non-radial systems, or interdependent systems.

The last category of solution examined by the survey was that of what the authors described as AI-based solutions: neural networks, genetic algorithms, and expert systems. An algorithm using a neural network was discussed because it sought to parallelize the system into zones which could be congifured seperately, but the system suffered from the same problems as any neural network, which is that they are only as good as the data provided, and they take time to train [16]. Only one method using a genetic algorithm was discussed, but it involved simply encoding switch states as an individual and using the objective function as the fitness function [17]. Similarly only one expert system was discussed, that by Taylor and Lubkeman [18], which acted as an extension of the heuristic proposed by Civanlar et. al. [13]. As before, none of these solutions recognized prioritization or interdependencies.

2.2 Service Restoration in Naval Shipboard Power Systems

-K.L. Butler-Purry, N.D.R. Sarma and I.V. Hicks

The goal of the authors of this paper was to create a method for prioritized reconfiguration of a shipboard power system [1]. To that end, they adapt the fixed-charge, network flow method to model an electrical utility system. In their method, the system is modeled as a network, where each component serves as an edge in a graph. Loads are nodes in the graph and are considered to be prioritized or nonprioritized, and either fixed or variable. Fixed loads can either be fully supplied or not supplied,

while variable loads can be partially supplied. Furthermore, paths to loads are also prioritized. For example, assuming two paths to a given load, a path which passes through more automatic transfer switches than the other (as opposed to passing through manual transfer switches) would have higher priority.

The loads and switches then are used as variables in forming a series of constraint equations and an objective function. The constraint equations cover aspects of the system such as current capacity and Kirchoff's Laws. The objective function then seeks to maximize the number of prioritized paths used and the value of the loads served. The value of a load is defined as a function of the load's capacity and a weighting factor based on whether or not the load is prioritized. Linear programming is then used to maximize the objective function. The authors finally proceed to show results of several simulations, both with and without a fault in the system, where this method found an optimal solution.

The proposed method has a number of advantages. It allows partial restoration of loads which can be partially restored, and it allows a limited qualitative evaluation of paths for loads which are fed from more than one path. These two factors allow greater flexibility in the possible reconfigurations of the system, and make it more likely that the ultimate solution will be optimal.

Unfortunately the authors' method also suffers from a number of limitations in terms of how it deals with prioritization. Priorities are a simple two-tiered system. Either a load is vital, or it is not. No allowance is made for a third load being somewhere between two others in importance. Furthermore, the assumption that loads will only have two paths is very restrictive, particularly for highly reconfigurable systems with multiple generators. The prioritization of paths becomes much more complex as the reconfigurability of the system increases, and the authors fail to account for that.

Lastly, the way the proposed method solves the linear program it creates requires holding a large matrix in memory. Each node requires at least five equations, and each edge requires one equation for

each phase, with more for systems with faults. For large systems, this can result in a very large matrix, and the calculations involved in solving it would be difficult to distribute across multiple processors.

2.3 Restoration of Services in Interdependent Infrastructure Systems: A Network Flows Approach

-E.E. Lee, J.E. Mitchell, and W.A. Wallace

The goal of Lee, Mitchell, and Wallace is significantly more ambitious than that of the others papers discussed here [4]. With this paper, the authors seek to provide a fully detailed mathematical model describing a set of critical infrastructure systems (electricity, water, etc.) and their interactions which could be used in creating a decision support system for system operators or analysts. It also supplies a detailed classification for the ways in which critical infrastructure systems can be interdependent, and described the use of their model in solving a problem in Manhattan.

The authors approach the problem from a network flows perspective. Each infrastructure system is considered to be its own network. Each component in each infrastructure system is represented by a node in the network, which can be supply nodes, demand nodes, or transhipment nodes. Each node has associated with it a supply and a demand, with supply nodes having no demand and demand nodes having no supply. Arcs represent connections between nodes, and have a capacity, a cost, and a flow. Equations are then developed which describe constraints such as maximum flow through a node and conservation of flow. Optimizing the system would then involve finding the min-cost flow.

To handle disruptions, the authors must expand the model beyond simple capacity and flow conservation constraints. First, the authors address intrasystem effects of a disruption by adding a slack variable s to each node, representing the node's demand shortfall. In addition, each node is also given a weight k, which serves as its priority. The authors then introduce system interdependencies to the model by introducing a binary variable y which represents the dependency between a demand node in one system and an arbitrary node in another system, and a variable b which represents the quantity of the service provided by that dependency. A value of 1 indicates that the demand of the demand node is

being supplied by the node in the other system, and a value of 0 indicates that demand is not met. The objective function for restoration is then based on minimizing a number of factors, primary among which are the sum of the products *ks* for each node in the system plus the sum of $kb(1-y)$ for each interdependency in the model, subject to constraints which are exhaustively detailed by the authors.

In addition to the extensive definition of terms involved with critical infrastructures and their interdependencies, the authors manage to provide an extremely detailed and complete model for critical infrastructure systems. It accounts for an arbitrary number of systems, it allows an arbitrary granularity in priorities, and it models system interdependencies. Because of the model's completeness, a solution achieved by solving it with a linear programming tool is very likely to be truly optimal.

The authors' model also has some flaws. As with the method proposed in [1], solving the system requires an extensive matrix calculation, which is again not easily distributable and thus relatively unsuited to real-time operations. The proposed model also only treats dependencies as being satisfied or unsatisfied. Though the authors concede that a dependency could in reality be partially satisfied, in the model they decline to allow for it.

The authors describe the use of their model and method of reconfiguration to solve a problem with infrastructure systems in Manhattan. While they are able to solve their model quickly, doing so requires a number of simplifications which greatly limit the precision and flexibility of their method. For example, in the power system all transformers along a feeder were aggregated into one transformer, as were all loads in a service area. Demand at the loads was also assumed to be at a given level for all loads in one of only two customer classes, and was further assumed to be constant. Making simplifications of this nature greatly speeds up solving an otherwise highly complex nonlinear system, but it reduces the precision of the flows that are found and decreases the possible granularity of potential solutions.

2.4 New Approach for Distribution Feeder Reconfiguration for Loss Reduction and Service Restoration

-W.M. Lin and H.C. Chin

The authors of this paper propose a reconfiguration method for radial systems [19]. In this paper, reconfiguration is split into two different problems: reconfiguration for optimization and reconfiguration for restoration. The goal of optimization is to improve a system which has suffered no service disruptions, while the goal of restoration is to have the system respond to address such a disruption.

The optimization algorithm starts by turning on all switches. For each loop created in this way, it then chooses a switch to turn off based on impedances in the loop, voltage losses across the loop, and distance from the tie switch in the loop. A tie switch is a switch which connects parts of the system which nominally are fed by different sources. The algorithm starts at loops closest to the sources in the system, and terminates once the system is radial.

The restoration algorithm turns on tie switches to restore service to an area where there has been a disruption based on the same criteria as in the optimization algorithm. If a constraint is violated by this action, the switch is turned back off and another is tried until either all choices are exhausted or one is found which works.

The proposed algorithms are simple and do not rely developing and solving a large matrix in order to arrive at a reconfiguration solution (though it is not explained how system voltages would be determined), which should result in better performance than some other solutions. However, the need for the restriction of the restoration algorithm to operating tie switches is not thoroughly explained. It is possible that a better solution could be found by allowing more switches inside the failed area to be operated, and the authors do not discuss whether the tradeoff between time and solution quality caused by increasing the search space was explored.

In addition, this method suffers from some of the same limitations as others previously discussed. There is no consideration of the relative importance of loads, and because the algorithms were proposed specifically for electrical systems there is no discussion of how to address interdependencies between infrastructure systems. The paper also suggests that failures in a system must be isolated from the rest of the system by operating switches bounding their segments, but seems to assume that this is done before reconfiguration begins. The algorithms are also restricted to radial systems, and so are inadequate for systems which operate with loops in order to service some loads.

2.5 A Heuristic Nonlinear Constructive Method for Distribution System Reconfiguration

-T.E. McDermott, I. Drezga, and R.P. Broadwater

Minimum Loss Optimization in Distribution Systems: Discrete Ascent Optimal Programming

-R.P. Broadwater, P.A. Dolloff, T.L. Herdman, R. Karamikhova, and A.F. Sargent

Together these papers describe an algorithm for reconfiguration called Discrete Ascent Optimal Programming (DAOP) that focuses on reducing line losses in the system [20,21]. DAOP considers the system being reconfigured as a graph where each component is an edge in the graph. Two subgraphs are considered: a supplied graph of all components receiving service and an unsupplied graph of all components not receiving service. The supplied graph is not necessarily connected, as mutiple sources could feed different parts of the entire system. The supplied and unsupplied graphs are mutually exclusive.

DAOP begins with all switches turned off such that almost the entire system is in the unsupplied graph. DAOP then searches for the source-ending load pair to restore which would result in the least loss increase. Ending loads are those loads in the unsupplied graph which are connected by a least-loss path to the supplied graph. DAOP then operates switches in order to restore the ending load in that pair. If a constraint is violated, the switches are reverted to their prior state. DAOP continues in this fashion until either all loads are restored or no more loads are restorable.

DAOP will always converge on a solution that is close to optimal in terms of reducing line losses, and does so without using matrix-based methods for determining the reconfiguration solution. In addition, it does not require that any simplifications or assumptions be made with regard to the system model. However it was not designed to take into account interdependent systems, and does not take into account load priority. Rather than try to restore the most important loads at any given time, DAOP is instead trying to restore the loads closest to the system area which is receiving service. While this may result in a good solution with respect to line losses, there is no guarantee that the solution will properly reflect the user's priorities.

2.6 Control Reconfiguration of Discrete Event Systems With Dynamic Control

-R. Sampath, H. Darabi, U. Buy, and J. Liu

The writers of this paper write about reconfiguration not for infrastructure systems, but rather for systems such as a hospital or a factory: a 'plant' with various stations where steps of a process happen [22]. While this prevents the work in this paper from being directly applicable to critical infrastructure reconfiguration, the principles are similar. In this paper, the authors describe a system as a 'plant' having a finite set of tasks to be accomplished, a set of constraints (i.e., available resources), 'places' in the plant, transitions between places, and controllers for the transitions. Places can consist of stations in the plant such as a hospital ER or a particular point on an assembly line, or they can represent a decision point in the next step to take such as the point at which a patient must be sent to recovery or the MRI. Each place has a number of transitions associated with it, each of which has exactly one output place. Together, these places and transitions form a directed graph. Each place also has a controller associated with it, which determines which transition to use at a given time based on the given constraints.

Reconfiguration in response to a disruption then involves updating the constraints on the controllers for the affected places, and checking if the new system state is feasible. If not, the algorithm then determines which sequence of transitions to take in order to restore normal operation. This is done with

a linear program that minimizes the cost of a sequence of transitions to reach an acceptable system state, the math for which is exhaustively detailed in the paper.

This method of reconfiguration is different from that required to examine critical infrastructure systems. For example, while the service provided by an infrastructure system could perhaps be modeled as flowing across transitions and system components as places, this would require modifying transitions to have mutiple outputs to allow for a component to feed mutiple other components. It also doesn't take into account the possibility of mutiple interdependent systems. Furthermore, while the matrix-based nature of the proposed algorithm is suitable for the relatively small example of modeling a hospital, it is unlikely to be as effective for large infrastructure systems which can feature thousands of components. While it may be possible to adapt this solution to reconfiguring critical infrastructure systems, it was not designed for that purpose and so doing so would be akin to forcing a square peg into a round hole.

2.7 Solutions Based on Genetic Algorithms and Other Evolutionary Techniques

A popular method for solving reconfiguration is to use some form of genetic algorithm to find a solution. In Augugliaro et. al. [23], Zhu [24], and Kumar et. al. [25] a simple genetic algorithm is used. Individuals in the populations of Zhu and Kumar et. al. are defined in terms of switch status (ON or OFF), while Augugliaro et. al. also includes capacitor banks and loads which can be disconnected. Crossover and mutation are used in all three, though different methods are used to ensure each individual remains a representation of a radial system. Augugliaro et. al. simply do not perform those operations on the parts of the individual representing switch status, while Zhu keeps a constant number of switches turned off and Kumar et. al. perform a radiality check on new crossovered/mutated individuals and force them back to radiality if that constraint is violated [23-25]. Augugliaro et. al. take the extra step of adding a 'branch exchange' operation which finds two switches in a loop and swaps their status.

Chang and Kuo [26] and Su and Lee [27] both use a genetic algorithm based on simulated annealing, in which an initial system state is iteratively 'moved' to neighboring states based on a fitness function (in both cases, the relative power loss between the two states, with lesser losses being preferred). At each move, a 'temperature' value is decreased, and the algorithm quits when the temperature reaches the freezing point.

Zhang et. al. [28] and Shin et. al. [29] propose solutions based on a TABU search (TS) algorithm. In a TS algorithm, the current state is moved to a neighboring state based on a fitness function and whether the potential new state is on a list of previously visited states. This is done using a mutation operation. States on this TABU list cannot be visited again, but as new states are added to the list old ones drop off. Zhang et. al. use a straightforward TS algorithm, while Shin et. al. combine a TABU list with a standard genetic algorithm to expand the number of possibilities being examined at once [28,29].

Another evolutionary technique used by Su et. al. [30] and Carpento and Chicco [31] is based on the pathfinding behavior of ants. In this kind of solution, a number of 'ants' are initialized to a given system state. The ants then iteratively move from system state to a neighboring system state. At each state, the ants deposit an amount of 'pheremone', which decays with each iteration. State transitions are determined by the amount of pheromone on candidate states and by problem-defined rules. Higher amounts of pheremone are preferred. In both Su et. al. and Carpento and Chico, the additional rules consider the relative power loss of the states under consideration. The algorithm terminates after a set number of generations or when all the ants reach the same state, whichever is first. The solution state is that which the greatest number of ants have found [30,31].

All of these evolutionary techniques are capable of finding a feasible solution to the reconfiguration problem, but they only examine electrical systems and with the exception of Kumar et. al., they all only seek to minimize power losses in the system. Kumar et. al. also seek to minimize switching operations

and number of loads dropped [25]. None of them address the prioritization of loads, nor do they account for interdependencies between system types. Furthermore, these kinds of solutions can take several minutes to run on even small systems, which likely makes them infeasible for the larger systems maintained by utility companies.

2.8 Market-based Multiagent System for Reconfiguration of Shipboard Power Systems

-K. Huang, S. Srivatava, D.A. Cartes, L.H. Sun

In this paper, Huang et.al. propose a multiagent system for reconfiguration of radial electrical systems in which each major component in the system is given its own agent [32]. These agents then coordinate with each other to determine how power should be routed through the system. Loads which have a higher priority in the system are granted their service requests first. This kind of solution has the advantage of decentralized control, and thus no single failure point. However, a distributed agent system like that described would require a lot of equipment in order to implement. Combined with the security concerns posed by the potential for rogue agents, an agent-based system like the one described could be cost-prohbitive to implement, particularly for large distribution systems. Furthermore, as with most other solutions, there is no accounting for system interdependencies.

2.9 Solutions based on Rules

Zhu et. al. [33] and Ding et. al. [34] propose methods for reconfiguration which make use of rules. Such solutions seek to decrease the potential solution space by use of a set of rules derived from a combination of past experiences and other knowledge of the system. In the case of Zhu et. al., these rules are enumerated in their paper and seek simply to eliminate potential switching operations which are likely to lead to either a greater power loss or a lesser power loss than another option. After applying each rule in sequence to a loop in the system, the resulting switch is operated. This is repeated

for each loop. This method is restricted to radial electrical systems and contains no accounting for load prioritization [33].

Ding et. al. propose a much more complex system, which recognizes the prioritization of loads in the context of missions the system must fullfill. Their solution involves three parts: a database, a load prioritization module, and an expert control actions module. The database contains information on the configuration of the system (including load status, switch states, bus transfer states, etc.) and the possible 'missions' in which the system may engage. The load prioritization module is responsible for generating a priority list of loads for the current mission configuration. It uses information about the loads such as their power factors and harmonic content, as well as the loads' importance to the current mission configuration, to generate the list of loads. This list is then provided to the expert control actions module. The expert control actions module then uses an expert rules system to evaluate the system configuration and the load priority list to determine which loads to shed and what operations to perform to accomplish the shedding [34].

The method proposed by Ding et. al. has the advantage of providing a great deal of flexibility. Among the factors involved in prioritizing loads are the results of expert interviews which help determine the relative importance of loads, and this rules-based method is capable of significantly pruning the number of switching options that must be examined [34]. Furthermore, the proposed method allows for mission changes to affect the relative priorities of loads in the system. This flexibility allows for a much greater chance of being able to find a solution for any given situation.

However, because the method proposed by Ding et. al. relies so much on expert interviews it has significant room for human error [34]. An erroneous assesment of the relative importance of various loads could result in the solution proposed by the algorithm being suboptimal. The mission priority lists used by this paper combine with its focus on load shedding also make it possible for this method to shed

low priority loads such that no mission is completely fulfilled. Furthermore, any model utilizing this method for reconfiguration will have to perform a large number of database accesses, which are quite expensive in terms of time and could prohibit this method from being used for large systems in an operational capacity.

2.10 Computer-based Strategy for the Restoration Problem in Electric Power Distribution Systems

-S. Ćurcić, C.S. Özveren, and K.L. Lo

The authors of this paper propose one of the few methods that recognize load prioritization as part of the reconfiguration problem [2]. Their method models the system as a graph in which components are nodes. The algorithm looks for 'islands' of unrestored load: connected subgraphs of the model with loads that are not receiving service. Their algorithm then proceeds through three stages for each island, addressing each island in order of decreasing size. It first tries to restore the island with a single switching operation. If that fails, the algorithm attemps to restore the island node-by-node, operating switches in decreasing order of spare capacity. Lastly, the algorithm will attempt to perform switching operations in order to eliminate bottlenecks – branches with a minimum available spare capacity – by looking for alternative sources for nodes supplied by a bottleneck.

Once all islands have been addressed in this way, the algorithm looks to see if there are any important loads which are not receiving supply. If so, then for each island with unrestored important loads, the algorithm tries to restore those important loads by disconnecting relatively unimportant loads, then tries to restore islands of unrestored load as describe above. Once all islands of unrestored important loads have been addressed, the algorithm terminates.

The proposed algorithm has the advantage of looking at the problem from the perspective of restoring prioritized loads, where most methods simply attempt to reduce losses. However, it seems to treat restoring important loads as something of an afterthought, rather than a primary goal of the

algorithm. Instead, most of the algorithm seems focused on just restoring as many loads as possible. In addition, the proposed method is only capable of handling radial electrical systems, and so doesn't take into account interdepenencies between different systems.

2.11 Optimized Restoration of Combined AC/DC Shipboard Power Systems Including Distributed Generation and Islanding Techniques

-S. Chushalani, J. Solanki, and N. Shulz

The authors of this paper present another of the few methods which recognize and take into account the prioritization of loads [3]. In the proposed method, the authors propose a method for the reconfiguration of a shipboard power system. The basic idea is to formulate constraint equations for the system, with an objective function to maximize the prioritized load restored. These constraints are then solved using a linear problem solver. The problem is split into two parts: restoration of balanced and restoration of unbalanced systems. In balanced systems, load in the system is balanced across all three phases, which is not the case for unbalanced systems. Consequently the constraints for balanced systems are much easier and more compact to express, which led the authors to handle the two problems separately.

The proposed solution does recognize prioritization, but only at three levels: vital, semi-vital, and non-vital. However, there is nothing intrinsic in the authors' formulation of the problem which would prohibit altering the method to allow for arbitrary priority levels. Despite having this advantage over other systems, the proposed method still suffers for relying on solving a large matrix (265 variables and 290 constraints for even a small unbalanced system) in order to reach a solution. Relying on matrices in this fashion is likely to cause a reconfiguration method to scale poorly, and thus be unsuitable for use with large distribution systems. Lastly, as is the case with almost all other solutions, the proposed method is restricted to electrical systems only, and does not account for system interdependencies.

2.12 Contributions

Past reconfiguration solutions all feature certain problems. Some do not account for load priority, reducing the likelyhood of an optimal solution. Some do not account for interdependencies among multiple system types, which limits the usefullness for large-scale, coordinated response. Many utilize matrices or linear programs to determine the reconfiguration solutions, which reduces the ability to respond rapidly to changes in the system. And some make simplifications to the models or flow algorithms which are likely to reduce the ability to find optimal solutions.

This dissertation seeks to address these problems by proposing a set of algorithms which may be used to reconfigure critical infrastructure systems. The proposed algorithms, based on the use of a Graph Trace Analysis (GTA) model, address all of the above problems. Each is run on a complete, unsimplified model, allows for arbitrary load prioritization, and accounts for the reliance of components from one system on components from another. Furthermore, they do not make use of matrix calculations to determine their solutions, and so they allow system parameters to be rapidly changed in the model and resolved without needing to completely reformulate the problem. The complexity of the proposed algorithms is also demonstrated to not be worse than polynomial in time with respect to the nature and size of the systems on which they are run.

Also proposed in this dissertation is a notation for describing algorithms for use with a GTA model. The GTA notation described in this paper is based on the Object Constraint Language, and uses sets and sequences in order to work with collections of components in the GTA model. This allows great flexibility when designing and implementing algorithms for GTA models. In addition, it is easy to implement algorithms using this notation with the features provided by the C++ Standard Template Library.

III. Graph Trace Analysis

Graph Trace Analysis (GTA) is a method of analyzing systems predicated on graphs [35]. GTA uses concepts of graphs; sets generated by tracing through the graph; where traces are implemented with iterators; and set operators. GTA uses a set of operations based on the Object Constraint Language described by Warmer and Kleppe [36]. The subset of GTA operators used in reconfiguration are described in Tables 1 and 2.

A GTA model is based on a multi-dimensional directed graph. Each component in a GTA model corresponds to an edge of the graph, and the nodes of the graph are the connections between components. Furthermore, the model of the graph is thought of just in terms of edges. That is, the model is an edge-edge graph, and nodes are not treated as separate entities, but become part of the edge itself. Edges in the model have the responsibility of tracking the other edges at their ends, but the connections themselves are not an explicit part of the model. Connectivity is therefore maintained among the edges of the GTA model by having each edge track the edges to which it is connected.

3.1 GTA Notation

The primary focus of GTA is on sets and sequences. While algorithms written with GTA notation do feature single-item variables, the operators provided by GTA are primarily intended to make working with collections of items easy. Operators specific to sets and sequences and members of GTA sets, sequences, and complex data structures are accessed using the → symbol. Sets are denoted by { } and sequences by []. Parts of an expression contained within parenthesis are executed before the rest of the expression.

The most important operator in GTA is the collect() operator. collect() operates on a set or a sequence, and takes as an argument an expression that evaluates to true or false for each element of the set or sequence in question. collect() then returns a collection consisting of all elements of the collection

a	b	Operation	Result	Effect
set or seq		a→size	int	The number of elements in a
set or seq	expr	a→collect(p| b)	set or seq	Returns all elements p in a for which b is true
seq		a→#index	int	The current position of the iteration index of a
seq	expr	a→iterate(b)	seq	Executes expression b on each element in a, with a→#index going from 0 to a→size-1. The result is a
seq	expr	a→riterate(b)	seq	Executes expression b on each element in a, with a→#index going from a→size-1 to 0. The result is a
set or seq	expr	a→order(b)	seq	Orders a such that its elements are in increasing order according to b. If a is a set, order makes it a seq
set or seq	any	a→includes(b)	bool	Returns whether b is an element of a
set or seq	any	a→excludes(b)	bool	Returns whether b is not an element of a
set or seq	any	a→including(b)	set	Returns the union of set a with element b
set or seq	any	a→excluding(b)	set or seq	Returns a except for element b. If a was a sequence, order is retained.
seq	any	a→prepend(b)	seq	Returns the sequence of b followed by the elements of a
seq	any	a→append(b)	seq	Returns the sequence of the elements of a followed by b
seq	int	a→at(b)	element	Returns the element of a at index b. 0 ≤ b < a→size
seq	seq	a→symmetricDifference(b)	seq	Returns the sequence of elements in either a or b but not both. First the elements of a in the order they were in in a, then the elements of b in the order they were in in b
set or seq	set or seq	a→intersection(b)	set	Returns the set of elements in both a and b
set or seq	expr	a→exists(b)	bool	Returns whether there is an element of a for which b is true
set or seq	expr	a→forall(b)	bool	Returns whether b is true for all elements of a
seq		a→first	element	The first element of a
seq		a→last	element	The last element of a
set or seq		max a	any	Returns the maximum element of a, as defined by the > operator. Requires that the elements of a have > defined.
set or seq	expr	a→sum(b)	any	Returns the sum of the expression b as applied to each element of a. Requires that + be defined for the elements of a.

Table 1: Collection-Specific GTA Operations Used In Reconfiguration

a	b	Operation	Result	Effect
any	any	a=b		Assigns b to a
any	any	a==b	bool	Returns whether a and b are equivalent
any	any	a<b	bool	Returns whether a is less than b. Works for any pair of data types for which < is defined
any	any	a>b	bool	Returns whether a is greater than b. Works for any pair of data types for which > is defined
any	any	a ≤ b,a ≥ b	bool	As > and <, but less than or equal to and greater than or equal to.
any	any	a +,-,*,/ b	any	Math operators. Work for any pair of data types for which they are defined.
bool	bool	a AND,OR b	bool	Boolean operators.

Table 2: Other GTA Operations Used In Reconfiguration

on which collect() was called for which the given expression is true. If the collection on which collect() is called is a set, so is the output collection. If the collection on which collect() is called is a sequence, then the output collection is a sequence where each element is in the same order as in in the input collection. For example, if sequence A=[1,2,3,4,5], then A→collect(p| p % 2 == 1) would return [1,3,5]. The collect() operator is a powerful tool in GTA for creating new sets and subsets of related objects, and is used extensively in the reconfiguration algorithms described in this dissertation.

3.2 GTA Traces

In a GTA model, each component has one and only one reference source. Though multiple sources can feed any given component, only one of those sources can be its reference source. The combination of the reference source and the graph topology in a GTA model defines a set of iterators for each component: forward, backward, feeder path, brother, and adjacent.

The forward and backward traces are used to trace through every component with the same reference source once and only once. The component in the forward trace from the current component and which has the same brother as the current component will receive flow from the current component, originating

with its reference source. The brother represents the first component in the forward trace of the current component not fed by the current component. Thus, once all components fed by the current component are included in the forward trace, the next component in the forward trace is the current component's brother. In this way, all components fed by the current component will be found in the forward trace before any components not fed by it. The backward trace is simply the reverse of the forward trace.

The feeder path trace for a given component gives the component which immediately feeds the given component. The feeder path trace is functionally complete, in that all other traces can be derived from it [35].

The adjacent trace gives a component physically connected to the current component but which is not in the forward or feeder path trace from the current component. An adjacent component may have a different reference source than the current component. The existence of an adjacent component marks a possible cotree location in the graph [35].

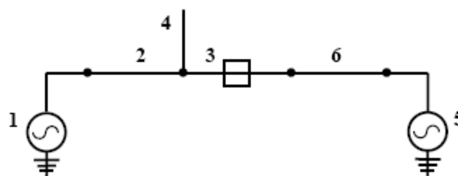

Figure 1: Simple GTA Model

As an example, Figure 1 shows a simple system with two sources and six components, where for purposes of discussion each component is identified with a unique number. In this system, the reference source of component 2 is 1. 1 is also the feeder path trace of 2. The forward trace component of 2 is 3. Component 3's forward trace is 4, which is also its brother trace. Component 3 also has an adjacent trace of 6.

Each of these traces defines an ordered sequence of components. These sequences are described in Figure 2. Thus, using the example in Figure 1, FT_3=[4], BT_3=[2, 1], FPT_3=[2, 1], BRT_3=[4], and ADJ_3=[6].

Sequences generated with traces can then be manipulated using the operators shown in Tables 1 and 2. The GTA trace sequences are described in Table 3.

FT_p = seq of components in forward trace from component p
BT_p = seq of components in backward trace from component p
FPT_p = seq of components in feeder path trace from component p
BRT_p = seq of components in brother trace from component p
ADJ_p = seq of components in adjacent trace from component p

Table 3: Seqs Created by GTA Traces

3.3 Component Structure for Reconfiguration and Problem Definition

Using GTA, the reconfiguration problem can be defined as follows. A component *C* is, for the purposes of reconfiguration, a 17-tuple which defines certain characteristics of the component important to reconfiguration.

C={p,type,systype,c,f,f_{req},ft,fpt,bt,brt,adjt,AD,OD,pri,status, $status_{dep}$, operable}
where p = unique component identifier
type = LOAD, SOURCE, SWITCH, OTHER
systype = system type: ELECTRIC, FLUID, GAS, and more as the user needs
c = capacity, or rating of a component
f = flow, where $f \leq c$
f_{req} = required flow if type == LOAD
ft,fpt,bt,brt,adjt = components related to C via forward, feeder path, backward, brother, and adjacent trace, respectively, where a value of 0 implies the component does not exist
AD = set of ‘AND’ dependencies
OD = set of ‘OR’ dependencies
pri = component priority
status = status of component-ON, OFF, FAILED
$status_{dep}$ = status of component’s dependencies
operable = whether the component can be turned on or off - YES, NO

These characteristics include the component type (specifically, whether or not it is a load, source, switch, or other), the system type (electrical, fluid, etc.), the flow capacity of the component c, the flow f through the component, the required flow f_{req} for the component, the component's trace information, the component's priority, and the components on which it is dependent. The adjacent trace for any component must have the same system type as that component.

A component's dependencies are contained in two sets, AD and OD. AD represents 'AND' dependencies, or dependencies which must all be satisfied for the current component to have service. OD represents 'OR' dependencies, or dependencies of which only one must be satisfied for the current component to have service. A dependency is a couplet of a component and a percentage. The component in the dependency couplet is the supporting component which must receive service to satisfy the dependency, and the percentage is the proportion of the supporting component's demand which must be met in order to satisfy the dependency. The percentage can be any value between 0 and 1, allowing for recognition of partial dependency requirements.

Dependency={pCmp, percent}
where pCmp = supporting component
percent = percent of supporting component's demand that must be met to satisfy the dependency

The $status_{dep}$ variable is a value indicating whether or not the component's dependencies are satisfied. If they are, $status_{dep}$ is 1. Otherwise, it is 0.

Figure 2 shows two situations with AND and OR dependencies. In Figure 2a, component 1 has AND dependencies on components 2 and 3. If either component 2 or component 3 is unrestored, then component 1's dependencies are unsatisfied. By contrast, in Figure 3b component 1 has OR dependencies on components 2 and 3. In Figure 2b, component 1's dependencies are satisfied as long as either or both of components 2 and 3 are restored.

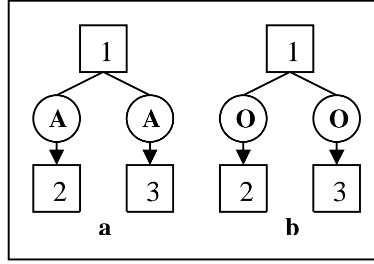

Figure 2: AND and OR Dependencies

A component is considered restored (status is ON) if its dependencies are satisfied and all components in its feeder path are restored. A component's dependencies are satisfied if all 'AND' dependencies are satisfied and at least one 'OR' dependency is satisfied. Table 4 illustrates GTA notation for a component with a status of ON where dependencies are satisfied.

Lastly, a switch component can be marked as operable or non-operable. For non-switchable components, C→operable is set to NO.

A GTA model M is a set of all components in the model. A system *S* is a subset of M. It is a collection of components such that for each component $C \in S$, *C*'s forward, backward, feeder path, brother, and adjacent trace are also contained in *S* (1). All components in a system must share a system type (C→systype is the same for all $C \in S$). Mutiple systems can be contained in a given model.

$$C^j \varepsilon S_j, S_j \rightarrow \text{includes}(C^j \rightarrow \text{ft}, C^j \rightarrow \text{bt}, C^j \rightarrow \text{fpt}, C^j \rightarrow \text{brt}, C^j \rightarrow \text{adjt}) \quad (1)$$

For a non-supporting component p (no component is dependent on p),
p→status==ON
implies $p \rightarrow f == p \rightarrow f_{req}$ AND $p \rightarrow status_{dep} == 1$ AND $FPT_p \rightarrow$forall(q→status==ON)

For a supporting component p with set of components ADeps with AND dependcies upon it and set of c components ODeps with OR dependencies upon it,
p→status==ON
implies ADeps→forall(p|$p \rightarrow f/p \rightarrow f_{req} \geq q$→AD→collect(r|r→pCmp==p) →first→percent) AND
ODeps→forall(p|$p \rightarrow f/p \rightarrow f_{req} \geq q$→OD→collect(r|r→pCmp==p) →first→percent) AND
$p \rightarrow status_{dep} == 1$ AND $FPT_p \rightarrow$forall(q|q→status==ON)

$p \rightarrow status_{dep} == 1$
iff p→AD→forall(q|q→pCmp→status==ON AND ($q \rightarrow pCmp \rightarrow f/q \rightarrow pCmp \rightarrow f_{req}) \geq q$→percent) AND
p→OD→exists(q|q→status==ON AND ($q \rightarrow pCmp \rightarrow f/q \rightarrow pCmp \rightarrow f_{req}) \geq q$→percent)

Table 4: Component Status Definitions

Two more subsets of components, the sources *Sources* and loads *Loads* are defined.

$$\text{Sources}=M \rightarrow \text{select}(p\rightarrow \text{type}==\text{SOURCE}) \quad (2)$$
$$\text{Loads}=M\rightarrow \text{select}(p\rightarrow \text{type}==\text{LOAD}) \quad (3)$$

Reconfiguration is then performed on the model M. When the status member of the component structure is mapped such that ON==1, OFF==0, and FAILED==0 and the boolean values TRUE and FALSE are mapped to 1 and 0 respectively, the objective of reconfiguration is to maximize the amount of load restored, weighted by the priority of the loads as follows:

$$\text{IsSupportingLoad}(p)=1- M\rightarrow \text{exists}(q|\ q\rightarrow AD\rightarrow \text{includes}(r|r\rightarrow pCmp==p)\ \text{OR}\ q\rightarrow OD\rightarrow \text{includes}(r|r\rightarrow pCmp==p)) \quad (4)$$

$$g = \max\{\ L\rightarrow \text{sum}(\ (p\rightarrow pri) * (1/(1 + |p\rightarrow f_{req} - p\rightarrow f|*\text{IsSupportingLoad}(p))) * (p\rightarrow status_{dep}) * (p\rightarrow status)\ )\ \} \quad (5)$$

In the objective function equation (5), a higher priority results in a higher value for a load’s term. The flow on the load being closer to its required flow also results in a higher value for that term unless the load is a supporting load as defined in (4), as supporting loads may only need to be partially supplied. Lastly, if the load’s dependencies are not satisfied or if it is not restored, the term drops out of the function completely.

The reconfiguration solution is subject to the constraint that no component in any system has a flow greater than its capacity:

$$M\rightarrow \text{collect}(p\rightarrow f \geq p\rightarrow c)\rightarrow \text{size} == 0 \quad (6)$$

This flow capacity constraint is not the only possible constraint to apply, but it is the only mandatory one as increasing flow beyond a component’s capacity can have disastrous consequences. The reconfiguration algorithms proposed in this dissertation are designed in a way that the implementation of them can include any number of additional constraint checks which may apply only to a specific system type or even a specific model.

As will be discussed in section 3.4, finding this optimal solution is NP-hard, and thus the algorithms presented in this dissertation attempt to solve a simplified version of the optimized reconfiguration

problem. The simplified problem is as follows. Given a model M as described in the optimized problem with set of switches S, find two sets of switches S_{open} and S_{closed} such that the following are true:

$$S_{open} \subset S, S_{closed} \subset S, S_{open} \cup S_{closed} = S, S_{open} \cap S_{closed} = \{\,\}, s \subset S_{open} \text{ } iff \text{ } s \rightarrow \text{status==OFF} \quad (7)$$

Furthermore, there must exist no switch $s \subset S$ such that changing the status of s increases the value of the non-maximized version of the function given in (5) and such that (6) is still satisfied. If one starts with all switches in either S_{open} or S_{closed}, then move switches one at a time to the other set, checking that the value of (5) increases and (6) is still satisfied each time, eventually this condition will be met. Since this operation requires O(sn) operations, this reduced problem is polynomial. However, so simple an algorithm ignores factors such as load prioritization and so more thorough methods such as those proposed in this dissertation are needed to arrive at a more optimal solution.

3.4 Complexity of Prioritized Reconfiguration

In order to show that prioritized reconfiguration is NP-hard, it must be shown that all NP-complete problems are reducible to it in polynomial time. Since all NP-complete problems are reducible to each other in polynomial time, it is sufficient to show that there is an NP-complete problem which is polynomial-time reducible to prioritized reconfiguration [37]. This means it must be shown that an arbitrary instance of a known NP-complete problem must have a polynomial-time mapping to an instance of prioritized reconfiguration.

The knapsack problem is a known NP-complete problem [38]. In the knapsack problem there is a knapsack of capacity *W*, and *n* items $x_{1..n}$ which have values $y_{1..n}$ and weights $w_{1..n}$. The objective of the knapsack problem is to fit the greatest value of items into the knapsack without going over its weight capacity:

$$\max \sum_{i=1}^{n} y_i x_i \text{ subject to } \sum_{i=1}^{n} w_i x_i \leq W \quad (8)$$

where $x_i = [0,1]$ indicates whether item *x* is or is not in the knapsack (1 is yes, 0 is no).

Theorem 1: Prioritized Reconfiguration is NP-hard

Let there be a knapsack with capacity W and n items $x_{1..n}$ with values $y_{1..n}$ and weights $w_{1..n}$.

Construct a system S as follows:

1. S contains n loads $L_{1..n}$ such that $L_i \rightarrow priority = y_i$ and $L_i \rightarrow f_{req} = w_i$.
2. Each load L_i in S is fed by a switch s_i with $s_i \rightarrow c \geq w_i$.
3. All switches s_i are fed by the same line l with $l \rightarrow c = W$.
4. Line l is fed by the system source R, where $R \rightarrow c = \sum_{i=1}^{n} w_i$.

The described system appears in Figure 3.

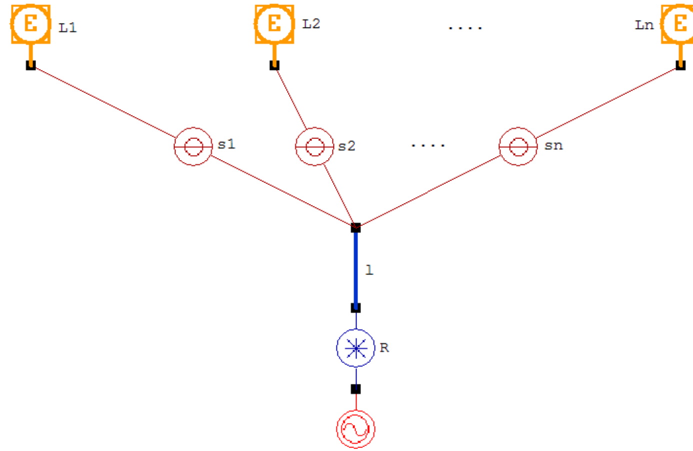

Figure 3: Constructed System *S*

In this constructed system S, if a switch s_i is turned on, the corresponding load L_i is restored and $L_i \rightarrow f$ is added to $l \rightarrow f$. By definition, if L_i is restored, $L_i \rightarrow status == 1$ and if L_i is not restored, $L_i \rightarrow status == 0$. Furthermore, because source R is capable of supplying enough flow for all loads at once, no load can be only partially fed ($L_i \rightarrow f < L_i \rightarrow f_{req}$ and $L_i \rightarrow status == 1$).

Thus, since S has no interdependencies, the reconfiguration problem for S is simply to operate switches in order to maximize:

$$\sum_{i=1}^{n} (L_i \rightarrow pri)(L_i \rightarrow status) \quad (9)$$

in accordance with (EQ#), subject to the constraint:

$$\sum_{i=1}^{n}(L_i \rightarrow f)(L_i \rightarrow status) \leq l \rightarrow c\ . \quad (10)$$

When the assigned values are substituted, the problem becomes that of maximizing:

$$\sum_{i=1}^{n}(y_i)(L_i \rightarrow status) \quad (11)$$

$$\text{subject to } \sum_{i=1}^{n}(w_i)(L_i \rightarrow status) \leq W \quad (12)$$

If a load being restored is considered equivalent to being placed in the knapsack, then the prioritized reconfiguration problem of the constructed system *S* is equivalent to the original knapsack problem.

Since the mapping from the knapsack problem to the prioritized reconfiguration problem involves a constant number of steps for each item x_i in the knapsack problem (creating the system objects and connecting them together in the prescribed fashion), this mapping is polynomial. Thus, prioritized reconfiguration must be at least as hard as the knapsack problem, and so prioritized reconfiguration is NP-hard.

What this does not mean is that reconfiguration is the same problem as the knapsack problem. Critical infrastructure systems are much more complex than the system representing an arbitrary knapsack problem shown in Figure 3, and system interdependencies add another layer of complexity missing from the knapsack problem definition. For this reason, it is better to develop a solution specifically for the reconfiguration problem as defined in Section 3.3, rather than try and apply a solution for the different and possibly less complex knapsack problem.

Finding an optimal solution for the reconfiguration problem requires considering the system as a whole. Because each component in a system has some maximum flow capacity, it is possible for the restoration of one load to prevent the restoration of other loads by consuming flow capacity on the components required to feed it. For example, a system may have three loads A, B, and C fed at some

point in their feeder paths by a single switch. This switch may only have enough capacity to feed either load A or both loads B and C. If A and B have the same priority, and C has a lesser priority, it may be optimal to restore B and C rather than A. This could only be determined by taking a global view of the system and considering the effects of restoring load A or load B on the ability to restore load C. Thus, finding an optimal solution for the reconfiguration problem requires considering what effect restoring one load will have on the ability to restore all other loads, in order to ensure that loads are maximally restored.

The algorithms presented in this paper take a more local view of the system, only considering a load or a segment at a time. Rather than trying to find an optimal solution to the problem as described in section 3.3, they focus on greedily trying to restore loads according to their priority level. The algorithms attempt to restore loads that have a higher priority (and hence higher value) before trying to restore lower priority loads. By doing so, the algorithms seek to get the most value out of the loads they do restore in the hopes that the resultant solution will be "good enough." In this way, they are able to determine a solution to the reconfiguration in polynomial time. The cost for this is that situations such as that described above can arise in which loads which are restored earlier in the process can cause the solution to be further from optimality than other solutions.

An additional problem caused by taking this local view of the system is that it becomes difficult for the algorithms to handle circular interdependency chains (A depends on B depends on C depends on A). Without being able to look at the system on a global level and determine a set of switch states which causes all loads in the circular interdependency chain to be restored, the algorithms presented in this dissertation can fail to restore some or all of them. The reason for this is that supporting loads must be restored before supported loads can be considered restored, and the way in which the algorithms choose to restore supporting loads may prevent them from later restoring the supported loads.

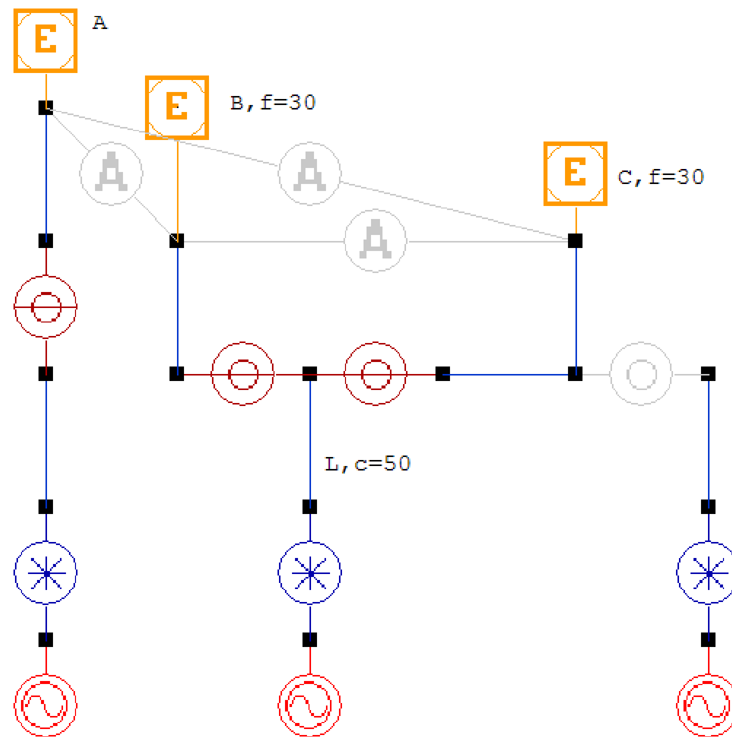

Figure 4:Circular Interdepenency Chain

Figure 4 shows such a circular interdependency chain. In the figure, Load A depends on Load B, which depends on Load C, which depends on Load A. Loads B and C each require a flow of 30, and line L has a maximum flow capacity of 50. If Load C is restored first, and it is restored through line L, then B will be unrestorable, as will A, and hence C will also be considered unrestorable. If the global view could be taken, it could be seen that load C could be restored through the switch on its right instead of the one on its left and that this would allow the entire chain to be restored.

This problem can also be seen in situations where there is no circular chain. In the system shown in Figure 4, even if Load A was not present (and hence the circular chain was broken), it would still be possible for a locally-focused algorithm to restore C through line L, preventing load B from being restorable.

3.5 Features of GTA

In a GTA model, changing the system by deleting or adding components only requires that local iterators be updated [35]. Thus any component can be found from any other component it interacts with via the various iterators. This simplifies the process of altering and maintaining a model because only local iterators need to be updated to reflect some topology change such as a switch operation.

Another primary consequence of using iterators to maintain graph topology is that GTA provides for a natural method of distributing the model and calculations on the model among processors. Because each component keeps track of the other components with iterators, it does not matter on which processor those other components are stored. Thus, it is straightforward to distribute the model across processors.

In addition, by standardizing the connections between components in this fashion, other qualities of components can be restricted to the components themselves. As a result, components can calculate values such as their own flows by examining their own qualities and iterators [35]. The consequence of this is that algorithms which must work with the whole model such as reconfiguration can abstract out system-specific factors by using GTA, and examine the interdependencies between multiple kinds of infrastructure systems at the same time.

IV. Reconfiguration Algorithms

This dissertation presents a collection of different algorithms for the prioritized reconfiguration of interdependent infrastructure systems described with GTA: From Loads, From Sources, Hybrid, and Cotree Switch. Each algorithm is run in two phases: one which tries to configure the system to be as radial as possible (with few or no loops), and one which adds loops as necessary. The first phase of the first three algorithms always result in a radially configured model, which is one in which there are no loops, while the first phase of the Cotree Switch algorithm is capable of resulting in a looped system. The second phase of each algorithm consists of an AddLoops function, which addresses underfed loads by adding loops to the model. Appendix A contains a full description of these algorithms written in GTA notation.

4.1 From Loads Algorithm

The From Loads Algorithm seeks to restore service by starting at the loads and working back toward the sources to develop a valid restoration path for each load. From Loads addresses the interdependent nature of multiple system types by recursively following dependencies on a potential restoration path and trying to find restoration paths for any supporting components in other systems. From Loads addresses priorities by attempting to restore loads in descending priority order. Thus, at any given time during execution there can be no restorable load with a higher priority than the one currently being restored. As a result of this approach, restoration of more important loads is never affected by restoration of less important loads.

From Loads initially turns off all switches. The loads in the model are then collected and sorted both by priority and such that all loads not in another component's dependency list occur before any loads that are in another component's dependency list. Thus, the sequence of loads will first contain all critical system loads before any supporting loads. This prevents the algorithm from restoring a

supporting load, which may only have a high priority because a critical load relies on it, before a lower priority critical load if that high priority critical load is unrestorable. At the same time, it allows the algorithm to later restore that supporting load if there is spare capacity after restoring critical loads. A component c's dependency list is the list of components in c→AD and c→OD. Restoration of each load is then attempted in this order.

```
Loads→order(p<q if (p→priority > q→priority OR
                        (M→exists(S|S→exists(t|t→AD→exists(u|u→ pCmp==q) OR
                                                  t→ OD→exists(u|u→ pCmp==q)))
                        AND
                        (M→ forall(S|S→ forall(t|t→ AD→collect(u|u→ pCmp==p) → size==0 AND
                                                  t→ OD→collect(u|u→ pCmp==p) → size==0)))))     (13)
```

For each load, if that load is unrestored, the algorithm collects a set of components via which the load could be connected to a source. That is, the feeder path traces from these components will lead to sources that could potentially supply service to the load. These components include the feeder path trace of the load, as well as any components with the same reference source as the load that have an adjacent trace. Let these components be placed in the set $Paths_l$, where l is the load of interest.

For each component c in $Paths_l$, starting with the load's feeder path trace, the algorithm creates a sequence $Path_{l\text{-}c}$ containing the path from l to a source through c. If the component c of $Paths_l$ being examined is the load l's feeder path trace, then $Path_{l\text{-}c}$ consists of the load l followed by l's feeder path. Otherwise, $Path_{l\text{-}c}$ consists of the load l, followed by the components connecting l and c, then c, then c→adjt, and finally c→adjt's feeder path.

Once $Path_{l\text{-}c}$ is created, the algorithm checks if it contains any components with status==FAILED. If so, $Path_{l\text{-}c}$ is not valid, and the algorithm proceeds to the next component in $Paths_l$. Otherwise, the algorithm develops a sequence named DecisionPoints from those in $Path_{l\text{-}c}$. The components in DecisionPoints are those components along $Path_{l\text{-}c}$ which must have some processing performed on

them during restoration. They consist of switches and components which have dependencies (AD or OD is not empty).

$$DecisionPoints=Path_{l\text{-}c}\rightarrow collect(p\rightarrow type==SWITCH\ OR\ p\rightarrow AD\ != \{\}\ OR\ p\rightarrow OD\ != \{\})\rightarrow reverse \quad (14)$$

Because $Path_{l\text{-}c}$ is a sequence ordered from l to c's reference source, DecisionPoints is ordered in the opposite fashion, such that the first element in DecisionPoints is the one closest to c's reference source. The algorithm addresses each one in turn, by operating switches or by recursively restoring components to satisfy dependencies. By addressing them in this order, service is gradually restored to the system rather than all at once. This allows problems restoring a given path would create, such as unsatisfiable dependencies, to be detected earlier in the restoration process, preventing the algorithm from wasting time.

Once l is restored, system constraints are checked for the components that have been affected by the restoration. These components are those which before restoration shared a reference source with either l or c. If any constraints are violated, the algorithm backs up along DecisionPoints looking for components which have OR dependencies (OD→size>0) and selects a different OR dependency to restore. If no set of OR dependencies of such components can be chosen such that l is restorable, the algorithm backs up its actions along DecisionPoints completely and selects the next component from $Paths_l$ to try. If no path is found that can restore l, it is deemed unrestorable and the algorithm moves on to the next load until all loads have either been restored or deemed unrestorable.

4.2 From Loads Example

Figure 5 shows a simple example system. The circuit on the left is a fluid circuit, and the other two are electrical circuits connected by an open switch. Components 6, 12, and 17 are loads. Load 6 has an assigned priority of 9, load 12 has an assigned priority of 5, and load 17 has an assigned priority of 6.

Load 6 has an AND dependency on load 17 represented by the ‘A’, which means load 17 must be restored in order for load 6 to be restored. Components 3, 7, 9, and 15 are switches. All other devices are transmission devices. For purposes of this simple example, there is enough capacity provided by the sources (1, 8, and 13) to service all of the loads, and the transmission devices are capable of handling all of the load present in their respective systems (electrical or fluid).

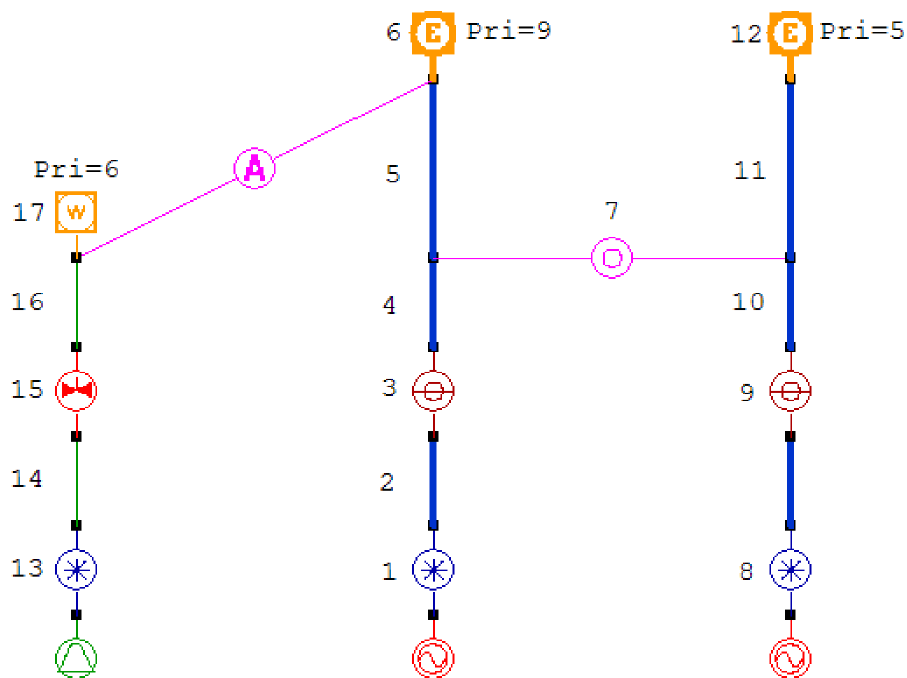

Figure 5: Sample GTA Model

From Loads begins by turning off the switches 3, 9, and 15. It then collects and sorts the loads as described in (12). For this system, Loads = [6,12, 17]. Note that although load 17 has an assigned priority greater than that of load 12, load 12 occurs first in Loads because load 17 is contained in load 6’s dependency list. From Loads then collects the potential paths for load 6. In this case, $Paths_6$=[5, 7]. From Loads first examines $Paths_{6-5}$=[6,5,4,3,2,1]. The decision points along $Paths_{6-5}$ are the switch at 3 and the load itself at 6 because it has a dependency. Thus, DecisionPoints=[3,6]. FromLoads then

addresses each element of DecisionPoints. Switch 3 is addressed by operating the switch (3→status=ON). Load 6 must be addressed by satisfying 6's dependencies: in this case by recursively restoring 6's AND dependency, load 17.

To restore load 17, From Loads collects $Paths_{17}$=[16], then DecisionPoints=[15]. The valve at 15 is operated (15→status=ON), at which point load 17 has been restored. Since execution is not at the top level of recursion, system constraints are not checked at this point. From Loads returns to the restoration of load 6. With load 17 restored, the dependencies of load 6 are satisfied, and so all of $Path_{6-5}$'s DecisionPoints have been successfully addressed, load 6 has been restored. System constraints are checked and no problems are discovered, so From Loads moves on to the next component in Loads, load 12.

Restoration of load 12 is similar to that of load 6, except that load 12 has no dependencies. $Paths_{12}$=[11,10], as 10→adjt==7. $Path_{12-11}$=[12,11,10,9,8], and so DecisionPoints=[9]. From Loads operates switch 9 (9→status=ON), at which point load 12 is restored. System constraints are checked and no problems are found. $Paths_{12-10}$ does not need to be checked, so From Loads next moves on to load 17. Since load 17 has already been restored, From Loads skips over it and finishes execution.

4.3 From Sources Algorithm

The From Sources algorithm approaches restoration from the opposite end of the system than the From Loads method. Rather than operating switches and satisfying dependencies while traversing from the load towards a source, this algorithm starts at the sources and works towards the loads. This algorithm has the potential for resulting in a more even distribution of loading among the sources. From Sources uses priority propagation from the loads back to the sources in order to determine which switches to operate at a given point. From Sources addresses system interdependencies by propagating priorities across dependencies. For example, if a priority 9 load is dependent on a component in another

system, that other component will be given a priority of at least 9. Further, at any given time, the algorithm will only attempt to turn on a switch with the highest priority from among its options, which ensures that higher priority loads will be restored before lower priority ones.

From Sources utilizes a “working priority.” The working priority is initially set to the highest priority present in the model, and the algorithm focuses on restoring loads with a priority at least equal to the working priority. As loads are restored, the working priority is gradually reduced.

The first step of From Sources is to turn off all switches. Next priorities are propagated from the loads back to the sources. For each component in a load’s feeder path, that component’s priority is set to the maximum of its own priority and that of the load. In addition, if that component has any dependencies, the component’s priority is recursively propagated down the feeder path of all components supporting it.

The algorithm then iteratively turns on switches for each source to expand the system area receiving service from that source. To decide which switch to turn on, the algorithm first collects all of each source’s bounding switches. The bounding switches of a source are those switches that have a status of OFF such that if they were turned ON, service would be provided to a segment of the system which is currently unserviced. The bounding switch set for a source is given by

$BoundSwts_{src}$ =
(FT_{src}→collect(p→type == SWITCH AND
p→status == OFF AND
p→fpt→status == ON))->including(FT_{src}→collect(p→adj| p→status == ON AND
p→adjt != 0 AND
p→adjt→type == SWITCH AND
p→adjt→status == OFF AND
p→adjt→fpt→status == OFF)) (15)

The algorithm then turns on one of these bounding switches with the highest priority. A number of checks are then performed on the system segment with restored service. These checks prevent the

algorithm from violating system constraints and also help to minimize the number of low-priority loads restored. Violations that could occur include:

- Flow constraint violations
- New segment contains a failed component

If any check fails, the switch is turned back off and the algorithm continues through the bounding switch list until it is exhausted or a valid switch is found.

A load is considered resolved once the algorithm has attempted to restore service to its segment by turning on a switch, whether or not it was successfully restored. Later operations may or may not restore a load that is initially not restorable as the algorithm continues execution.

Once a sectionalizing device has been turned on or the algorithm determines there are no possible on switch operations, an accounting is made of the resolved loads. If all loads with priority greater than or equal to the working priority have been resolved, the working priority is set to the next lowest priority and the algorithm repeats the described steps until either all loads are restored or no switches can be successfully turned on. Priorities are also repropagated to account for the changed topology of the model.

4.4 From Sources Example

Using the same example model as before (Figure 5), From Sources begins by turning off the switches 3, 9, and 15. It then collects and sorts the loads by priority, so that Loads = [6,17,12]. From Sources then collects the sources in the system and their boundary switches. In this case Sources = [{1, {{3, false}}}, {8, {{9, false}}}, {13, {{15, false}}}]. The first element in each of the couplets in the Sources sequence is the source component (1, 8, and 13). The second element is the set of boundary switches paired with whether From Sources has yet tried to operate that switch. The latter value is initialized to false. The boundary switch for source 1 is 3, for source 8 it is 9, and for source 13 it is 15.

From Sources also propagates priorities back from the loads and across dependencies. Note that this changes load 17's priority to 9. The working priority is set to 9.

From Sources then sorts Sources in increasing order of the proportion of their capacity being used (p→f/p→c). Since there is currently no flow on any source, the order of Sources is not changed. The next step is to iterate across Sources and attempt to close a boundary switch for each source. For source 1, switch 3 is turned on because it has a priority of 9, and constraints are checked. Since there are no problems, 3's boundary switches are updated to {{7, false}}, and priorities are repropagated. From Sources then moves on to source 8, but does not find any switches to operate because 9→priority == 5, which is lower than the working priority. From Sources completes this iteration across Sources by turning on switch 15 for source 13 and updates 13's boundary switches to {}.

At this point all loads with a priority greater than or equal to the working priority have had service restored to them (loads 6 and 17). Thus, From Sources updates the working priority to the next lowest priority, which in this case is 5, the priority of load 12. The above steps are repeated. Sources is sorted, but because source 8 has no flow on it, Sources = [{8, {{9, false}}, {1, {{7, false}}, {13, {}}]. From Sources iterates across sources, operating switch 9 for source 8 and restoring service to load 12. Constraints are checked and there are no problems, so 8's boundary switches are updated to {{7, false}}. Also at this point, all loads with priority greater than or equal to the working priority have been addressed, so the working priority is updated to 0, as there are no unaddressed loads left with a higher priority.

When From Sources considers the boundary switches for source 1, it now finds none that can be operated because both sides of switch 7 have service. 1's boundary switches are thus updated to {{7, true}}. Since all of source 1's boundary switches have been visited, but no operable ones found, From

Sources moves on to source 13 and again finds no operable switches. From Sources iterates through the sources one more time, but finds no switches than can be operated, and so the algorithm terminates.

4.5 Hybrid Algorithm

The Hybrid algorithm combines aspects of the From Sources and From Loads algorithms. Like the From Sources algorithm, it approaches the problem from the sources, turning on switches as it spreads towards the loads. Unlike the From Sources method, the operable devices are limited according to those which could be used to restore the unresolved loads of the current highest priority. This adds some initial overhead, but places stronger restrictions on which switches can be operated, limiting the number of cases in which lower priority loads get restored before higher priority loads. Again, system interdependencies are addressed through the same priority propagation as they are in the From Sources method.

After all switches are turned off and priorities have been propagated (as in From Sources), the algorithm creates the set of potential feeder paths $Paths_l$ for each load of the current highest priority as in the From Loads method. Each switch along the paths $Path_{l\text{-}c}$ as described in the From Loads method is marked as operable. The algorithm then proceeds as it does in From Sources, except that it does not attempt to turn on any devices not marked as operable. When all of the loads at the working priority level have been resolved as in From Sources, the algorithm marks the devices on potential feeder paths to loads of the next highest priority as operable.

In this way, by restoring service to the system starting from the sources but only allowing operation of devices that could feed loads of highest priority at a given time, the Hybrid method keeps the forced balance of the From Sources method while further minimizing the number of low priority loads restored at the expense of high priority loads.

4.6 Hybrid Example

Again using the system in Figure 5, Hybrid begins by turning off the switches 3, 9, and 15, then marks all of those switches inoperable (p→operable=FALSE). It then collects and sorts the loads by priority. As in From Sources, Loads = [6,17, 12]. The working priority is then set at 9, which is the priority of the highest priority load and priorities are propagated back from the loads, which changes 17's priority to 9.

Hybrid then collects the potential paths for load 6 in order to make operable those switches along paths which could feed it. In this case, $Paths_{6}$=[5, 7]. Hybrid first examines $Paths_{6-5}$=[6,5,4,3,2,1]. Since 6 has a dependency, Hybrid goes across that dependency to make operable the switches feeding the supporting load 17. $Paths_{17}$=[16], so $Paths_{17-16}$=[17,16,15,14,13]. The only switch along $Paths_{17-16}$ is 15, so Hybrid sets it operable (p→operable=TRUE) and returns to processing $Paths_{6-5}$. The only switch along $Paths_{6-5}$ is component 3, and it is set to be operable. Hybrid moves on to the next load at or above the working priority, 17, but since 17 has already been addressed it skips it. No other loads are at or above the working priority, so Hybrid moves on to restoring loads.

Hybrid next collects the sources in the system and their boundary switches. In this case Sources = [{1, {{3, false}}}, {8, {{9, false}}}, {13, {{15, false}}}]. The first element in each of the couplets in the Sources sequence is the source component (1, 8, and 13). The second element is the set of boundary switches paired with whether Hybrid has yet tried to operate that switch. The latter value is initialized to false. The boundary switch for source 1 is 3, for source 8 it is 9, and for source 13 it is 15.

Hybrid then sorts Sources in increasing order of the proportion of their capacity being used (p→f/p→c). Since there is currently no flow on any source, the order of Sources is not changed. The next step is to iterate across Sources and attempt to close a boundary switch for each source. For source 1, switch 3 is turned on because it has a priority of 9 and is operable, and constraints are checked. Since

there are no problems, 3's boundary switches are updated to {{7, false}}, and priorities are repropagated. Hybrid then moves on to source 8, but does not find any switches to operate because switch 9 is not operable. Hybrid completes this iteration across Sources by turning on switch 15 for source 13 and updates 13's boundary switches to {}.

At this point all loads with a priority greater than or equal to the working priority have had service restored to them (loads 6 and 17). Thus, Hybrid updates the working priority to the next lowest priority, which in this case is 5, the priority of load 12. With a new working priority set, Hybrid attempts to mark more switches as operable. Loads 6 and 17 have been restored, so they are ignored, but load 12 has not. $Paths_{12}$=[11], so Hybrid sets as operable all switches along $Paths_{12\text{-}11}$=[12,11,10,9,8,7], which in this case is just component 9.

The above steps are repeated. Sources is sorted, but because source 8 has no flow on it, Sources = [{8, {{9, false}}, {1, {{7, false}}, {13, {}}]. Hybrid iterates across sources, operating switch 9 for source 8 and restoring service to load 12. Constraints are checked and there are no problems, so 8's boundary switches are updated to {{7, false}}. Also at this point, all loads with priority greater than or equal to the working priority have been addressed, so the working priority is updated to 0, as there are no unaddressed loads left with a higher priority.

When Hybrid considers the boundary switches for source 1, it now finds none that can be operated because both sides of switch 7 have service. 1's boundary switches are thus updated to {{7, true}}. Since all of source 1's boundary switches have been visited, but no operable ones found, From Sources moves on to source 13 and again finds no operable switches. Hybrid iterates through the sources one more time, but finds no switches than can be operated, and so the algorithm terminates.

4.7 Cotree Switch Algorithm

The Cotree Switch method attempts to minimize the number of operations which must be performed in order to successfully reconfigure the system(s). The Cotree Switch algorithm is similar to those described in [2], but does not share the same limitations. Unlike the other algorithms proposed here, it starts by turning on all switches. Priorities are propagated from the loads as in From Sources, and then any failed components in the system are isolated by turning off the switches bounding the failures' respective segments. As in From Sources and Hybrid, system interdependencies are addressed by propagating priorities across dependencies.

After switches are closed and failures isolated, a system constraint check is then performed. If there are any constraint violations, the components where the violations occur are sorted by increasing priority. For each violation, the algorithm collects all components feeding and fed by the violation into a set $FullPath_{viol}$ as described in (16). $FullPath_{viol}$ is then searched for the turned-on switch with the lowest priority, as described in (17). That switch is turned off, priorities are repropagated, and system constraints are checked again. Because turning switches off drops loads, this gradually reduces the flow on the system and thus eliminates constraint violations. This is repeated until there are no more constraint violations. Using GTA notation this can be described as follows for a given component with a violation viol.

$$FullPath_{viol}=(FPT_{viol}\rightarrow append(FT_{viol}\rightarrow collect(\ p\rightarrow ft\ !=\ viol\rightarrow brt\ AND\ FPT_{p}\rightarrow includes(viol\rightarrow brt) == false)\ ) \rightarrow prepend(viol) \quad (16)$$

$$OperateDevice_{viol}=FullPath_{viol}\rightarrow collect(p|\ p\rightarrow type==SWITCH\ AND\ p\rightarrow status==ON\ AND\ FullPath_{viol}\rightarrow includes(q\rightarrow pri<p\rightarrow pri)==false)\rightarrow first \quad (17)$$

Once all constraint violations have been eliminated, the algorithm collects all of the cotrees in the system(s). Cotrees are switches which are turned on and create independent loops in the system. They

can be found by collecting those switches which are turned on and have an adjacent component which is also receiving service, as described in (18).

$$Cotrees_S = S \rightarrow collect(p \rightarrow type == SWITCH\ AND\ p \rightarrow status == ON\ AND\ p \rightarrow adjt \rightarrow status == ON) \quad (18)$$

For each Cotree, the algorithm creates a set of the switches along the feeder path of the cotree device and its adjacent trace. These devices are then sorted by flow, and the one with the least flow is turned off. A constraint check is performed, and if there is a violation or if a load has become unrestored, the sectionalizing device is turned back on. The algorithm terminates once each cotree has been addressed in this manner.

4.8 Cotree Switch Example

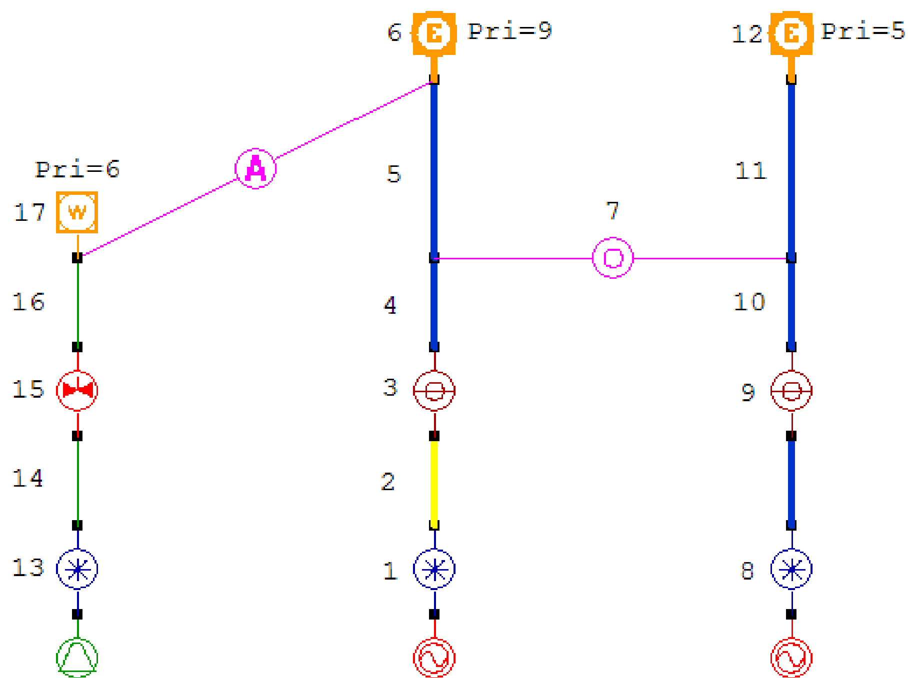

Figure 6: Sample GTA Model With Failure

In Figure 6, component 2 of the sample model has suffered a failure when Cotree Switch starts to run. Cotree Switch starts by turning on all switches, which in this case means closing switch 7 since all others are already on. This creates a cotree between component 7 and component 10. Cotree Switch then collects the model loads into Loads=[6,12,17], all failed components into a sequence Failures, and also a sequence Violations of all failed components and all components C where $C \rightarrow f > C \rightarrow c$. In this model, Failures=[2] and Violations=[]. Loads is then ordered as in From Loads, so Loads=[6,12,17], then priorities are propagated back from the loads.

Cotree Switch then operates switches to isolate those components in Failures. In this case, the only failure is at component 2. Failures are isolated by turning off all switches bordering that component's segment, and in this case the only such switch is component 3. Turning off switch 3 removes the cotree added earlier. Cotree Switch now moves on to remedying constraint violations in the system by opening low priority switches fed by the violating components. Since Violations is empty for this model, Cotree Switches skips this step.

Lastly, Cotree Switches tries to turn off switches in order to eliminate loops that have been created in the system. Again, turning off switch 3 removed the only loop in the system, so Cotree Switch skips this step and terminates. After Cotree Switch executes, load 6 has ceased to be fed through a faulty line by source 1, and is instead fed through switch 7 on an alternate route by source 8.

4.9 Adding Loops

Once any of the above algorithms have terminated, reconfiguration must look to see if any of the loads to which service has been restored are receiving sufficient service. Because it is possible for a single source not to be able to provide enough flow to a load to meet its demand, reconfiguration uses an Add Loops algorithm to find any such loads and further operate switches in order to increase the flow that can reach the load, creating independent loops in the model. An additional consideration is the

possibility that a component may have a dependency, but may not require the component on which it is dependent to be fully served in order for that dependency to be satisfied. Add Loops also accounts for partial dependencies, and does not attempt to provide more power to supporting loads which already have met the demand placed on them by the component they support.

The first step in adding loops is to collect all loads that are underfed as described in (19), and sort them as in From Loads. Underfed loads are those loads which have flow less than their required flow, or any load L which is part of another component C's AND depenency list such that the flow on L does not meet the demand required by C, or any load L which is part of a component C's OR dependency list such that none of C's OR dependencies are sufficiently met. These loads are held in a sequence Underfed.

Thus Load l in Model M is underfed if and only if:

$$
\begin{aligned}
&l\rightarrow f < l\rightarrow f_{req} \text{ OR} \\
&M\rightarrow exists(p|\ p\rightarrow AD\rightarrow exists(q|\ q\rightarrow pCmp==l \text{ AND } l\rightarrow f/l\rightarrow f_{req} < q\rightarrow percent)) \text{ OR} \\
&M\rightarrow exists(p|\ p\rightarrow OD\rightarrow exists(q|\ q\rightarrow pCmp==l \text{ AND } l\rightarrow f/l\rightarrow f_{req} < q\rightarrow percent) \text{ AND} \\
&\qquad p\rightarrow OD\rightarrow exists(q|\ q\rightarrow pCmp\rightarrow f/q\rightarrow pCmp\rightarrow f_{req} > q\rightarrow percent)==FALSE) \qquad (19)
\end{aligned}
$$

Once the underfed loads are collected and sorted, Add Loops iterates across each load l in Underfed. If l is still underfed, then for each component c in FPT_l, Add Loops collects a set of components SegSwts consisting of all switches already in SegSwts, all switches in c's segment that are off, and all components with an adjacent trace in c's segment which are switches that are off (20).

$$
\begin{aligned}
&SegSwts=SegSwts\rightarrow including(Segment_c\rightarrow collect(p|\ p\rightarrow type==SWITCH \text{ AND } p\rightarrow status==OFF) \\
&\qquad\rightarrow including(Segment_c\rightarrow collect(p\rightarrow adjt|\ p\rightarrow adjt!=NULL \text{ AND} \\
&\qquad\qquad p\rightarrow adjt\rightarrow type==SWITCH \text{ AND} \\
&\qquad\qquad p\rightarrow adjt\rightarrow status==OFF))) \qquad (20)
\end{aligned}
$$

Add Loops then turns on all switches in SegSwts. System constraints are checked, and if there are any, all switches in SegSwts are reopened and Add Loops gives up on l. If there are no system constraint violations and l is no longer underfed, Add Loads sorts SegSwts in order of increasing flow

on the switches, and opens them until it cannot open more without causing l to be underfed. Add loops then moves on to the next load in Underfed.

If after all switches in SegSwts are closed there are no constraint violations but l is still underfed, Add Loops moves on to the next component in FPT_l not in the same segment as c, and repeats the above for that segment. Add Loops continues this until l is either completely fed or there are no more switches to close to further supply it.

Add Loops terminates once all loads in Underfed have been addressed in this fashion.

V. Algorithm Performance

5.1 Example 1: Real-World Electrical Model

To examine the performance of these algorithms, an implementation of each was run on a model of a real distribution system and compared to an implementation of the Discrete Ascent Optimal Programming (DAOP) algorithm described by Broadwater et. al. [17,18]. Tests were performed using a 2.00GHz Intel Pentium M processor. The model used in testing is shown in Figure 7 It contained 1835 load points and seven sources.

Figure 7: Real-World Electrical Model

The yellow components in Figure 7 indicate components which have failed, dropping several hundred of the 1,835 loads, with the exact number based on which switches are allowed to be operated. Switching operations were restricted to three-phase devices, of which there were 171. Each algorithm (except for DAOP, which did not provide this capability) was run in both one and two stages. In the two-stage runs, the first stage only allowed operation of major, automatic 3-phase devices, while the second stage allowed operation of all 3-phase devices. In the runs with the DAOP algorithm all 3 phase

switches were operable. The first stage also only used a relatively quick load estimation technique for judging constraint violations, while the second stage used a full non-linear power flow. The purpose of the two-stage process was to emulate what might be done in the even of a real emergency: performing a fast reconfiguration with devices that could be operated remotely for an immediate response, followed by a slower and more thorough run using all devices to determine what should best be done manually.

Tables 5-11 show the results of the runs. All of the algorithms proposed in this dissertation were able to run in under 18s in the single-stage runs and under 30s in the 2-stage runs, configuring the system to provide service to 98.96% of the load points in all cases. In comparison, the DAOP algorithm took over five and a half minutes to run, and was only able to provide service to 86.1% of load points. Even just the first stage of the 2-stage runs manage to restore a substantially higher percentage of load points than DAOP.

Time	From Loads	From Sources	Hybrid	Cotree Switch	DAOP
1 Stage	7.68s	9.30s	17.55s	2.67s	335s
2 Stages	2.30s	1.72s	4.00s	1.64s	
	7.55s	9.51s	25.27s	5.65s	

Table 5: Ex. 1 Algorithm Comparison in Time

Amount Serviced	From Loads	From Sources	Hybrid	Cotree Switch	DAOP
1 Stage	98.96%	98.96%	98.96%	98.96%	86.10%
2 Stages	95.00%	95.00%	95.00%	94.99%	
	98.96%	98.96%	98.96%	98.96%	

Table 6: Ex. 1 Algorithm Comparison in Loads Serviced

Mean Phase Imbalance	From Loads	From Sources	Hybrid	Cotree Switch	DAOP
1 Stage	21.71%	19.86%	16.57%	8.71%	17.29%
2 Stages	26.43%	25.29%	25.29%	18.57%	
	21.71%	18.86%	16.86%	8.71%	

Table 7: Ex. 1 Algorithm Comparison in Mean Phase Imbalance Across Sources

Switches Operated	From Loads	From Sources	Hybrid	Cotree Switch	DAOP
1 Stage	11	13	21	15	66
2 Stages	3	9	9	9	
	10(1)	12(4)	21(4)	6(1)	

Table 8: Ex. 1 Algorithm Comparison in Number of Switches Operated

kW Losses	From Loads	From Sources	Hybrid	Cotree Switch	DAOP
1 Stage	20.34	20.91	20.93	18.69	25.47
2 Stages	20.16	20.99	20.99	19.03	
	20.34	21.00	20.91	18.69	

Table 9: Ex. 1 Algorithm Comparison in Mean kW Losses Across Sources and Phases

Mean kW Flow	From Loads	From Sources	Hybrid	Cotree Switch	DAOP
1 Stage	3231	3250	3249	3244	2758
2 Stages	3140	3148	3148	3155	
	3231	3250	3249	3244	

Table 10: Ex. 1 Algorithm Comparison in Mean kW Flow Per Source

kW Flow σ	From Loads	From Sources	Hybrid	Cotree Switch	DAOP
1 Stage	1774	1539	2248	1981	3517
2 Stages	1873	2100	2100	2344	
	1774	1557	2244	1981	

Table 11: Ex. 1 Algorithm Comparison in Standard Deviation of kW Across All Sources

Table 7 shows mean phase imbalance across the loads in the system. By this measurement, DAOP falls in the middle of the field, with the Hybrid and Cotree Switch algorithms producing smaller phase imbalances and the From Loads and From Sources methods providing larger ones. While the first stage of the two-stage runs is unambiguously worse than DAOP by this metric in all cases, the first stage solution is not meant to last longer than it takes to get crews out to perform the second stage reconfiguration.

Table 8 shows the number of switches operated for each run. For the two-stage runs, the number in parentheses is the switches operated in stage 1 which were returned to their original state in stage 2. All four algorithms proposed in this dissertation perform much better than DAOP at this metric, requiring far fewer switches to be operated in order to reach their solution states, with From Loads and From Sources being the top performers in the single stage run and Cotree Switch being the best in the two stage run.

Table 9 shows the losses in kW resulting from each system configuration generated by the various algorithms. To determine these figures, the losses per source per phase were averaged. By this metric, DAOP, which was designed with the minimization of losses in mind [20,21], performs between 21.7% and 36.3% worse than all four proposed algorithms in the single-stage runs, and between 21.3% and 25.2% worse than the complete two-stage runs (and a little more worse than the first stage of two of those runs). The algorithms proposed in this dissertation are all remarkably consistent in the losses their solutions allow on this system, ranging in a narrow band from 18.69kW to 20.99kW.

Lastly, Tables 10 and 11 show information regarding the distribution of kW flow across the sources in the system. The mean flow for each of the proposed algorithms shown in Table 10 exists within a very narrow range, as they all restore the same amount of load, while the DAOP algorithm has a much lower mean reflecting its much lower amount of load restored. However, in Table 11 it is apparent that the proposed algorithms all produce a much more even distribution of load across the sources than DAOP. The best performer by this metric is the From Sources method, which was specifically designed to try and create a better load balance.

5.2 Example 2: Large Real-World Electrical Model

Figure 8 shows a real-world electrical model that is much larger than that shown in Figure 7, containing 961 three-phase switches and 9,598 loads. As before, each algorithm described in this dissertation was run on the model in both one and two stages. Attempts were made to run DAOP on the model as well, but DAOP was unable to reach a solution for the model. As with the first example, the yellow components in the upper left are those which have failed.

Tables 12-14 show the results of the algorithm runs. In these runs, the Cotree Switch algorithm is clearly the better solution when it comes to execution time and loads restored, but falls short of the From Sources algorithm in terms of number of switches operated. Interestingly, the first stage of the two stage

runs for From Loads, From Sources, and Hybrid restores more loads than the second stage. This is attributable to the different constraint check methods utilized in the two stages, with the first stage only performing a flow estimate as opposed to the more exacting full power flow algorithm used in the second stage.

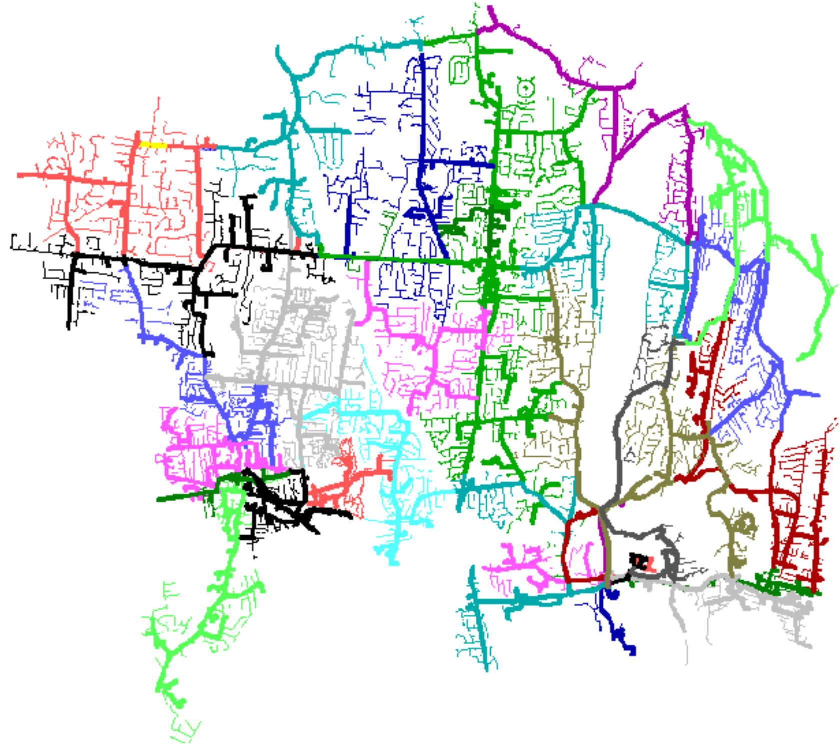

Figure 8: Large Real-World Electrical Model

Time	From Loads	From Sources	Hybrid	Cotree Switch	DAOP
1 Stage	2m47s	6m1s	6m36s	2m23s	----
2 Stages	1m27s	1m37s	1m52s	1m34s	
	3m3s	6m30s	8m15s	5m1s	

Table 12: Ex. 2 Algorithm Comparison in Time

<table>
<tr><th>Amount Serviced</th><th>From Loads</th><th>From Sources</th><th>Hybrid</th><th>Cotree Switch</th><th>DAOP</th></tr>
<tr><td>1 Stage</td><td>97.98%</td><td>97.97%</td><td>97.96%</td><td>99.52%</td><td rowspan="3">----</td></tr>
<tr><td rowspan="2">2 Stages</td><td>99.34%</td><td>99.34%</td><td>99.34%</td><td>99.34%</td></tr>
<tr><td>97.98%</td><td>97.98%</td><td>97.96%</td><td>99.52%</td></tr>
</table>

Table 13: Ex. 2 Algorithm Comparison in Loads Restored

<table>
<tr><th>Switches Operated</th><th>From Loads</th><th>From Sources</th><th>Hybrid</th><th>Cotree Switch</th><th>DAOP</th></tr>
<tr><td>1 Stage</td><td>99</td><td>73</td><td>129</td><td>86</td><td rowspan="3">----</td></tr>
<tr><td rowspan="2">2 Stages</td><td>3</td><td>19</td><td>25</td><td>50</td></tr>
<tr><td>98(1)</td><td>68(10)</td><td>146(15)</td><td>38(1)</td></tr>
</table>

Table 14: Ex. 2 Algorithm Comparison in Switches Operated

5.3 Example 3: Integrated Model

The proposed methods were also run in one stage on the system shown in Figure 9. This system contains both electrical and fluid circuits, as well as a number of logical loads defining system missions. Missions are represented by the red squares and are dependent on loads in the electrical and fluid systems. The fluid system is represented by the green lines and the electrical system by the black and brown ones. The yellow line represents a component which has failed. The missions AAW (Anti-Air Warfare, priority 5) and ASW (Anti-Surface Warfare, priority 3) have OR dependencies on Radar 2, Gun, and Radar 1, each of which have at least one AND dependency on one or more of the physical circuits. The three propulsion missions (from left to right, priorities 9, 2, and 6) have AND dependencies on electrical loads, and the two pumps in the fluid circuits each have an AND dependency on an electrical load. There are five independent electrical loads (those on which no mission is dependent) with priorities of 9, 0, 0, 0, and 0. A number of the loads are grouped into 2 ‘panels’ fed by automatic transfer switches.

Figures 10a-10d show the results of running the proposed methods on the system in Figure 9 if the line feeding the right panel is failed. Components which have become pink have lost service.The From Sources and Hybrid methods yielded the same result. Both of them restore all loads except for the large priority 0 load on the lower right of the left panel. The From Loads algorithm does not restore Radar 2,

because doing so was unnecessary to restore AAW. The result of ignoring the loads supporting Radar 2 is that the large load on the left panel was restorable. Finally, the Cotree Switch method manages to restore all loads, but only does so by creating a loop in the electrical system. The Cotree Switch method is the only method which can create loops in this way.

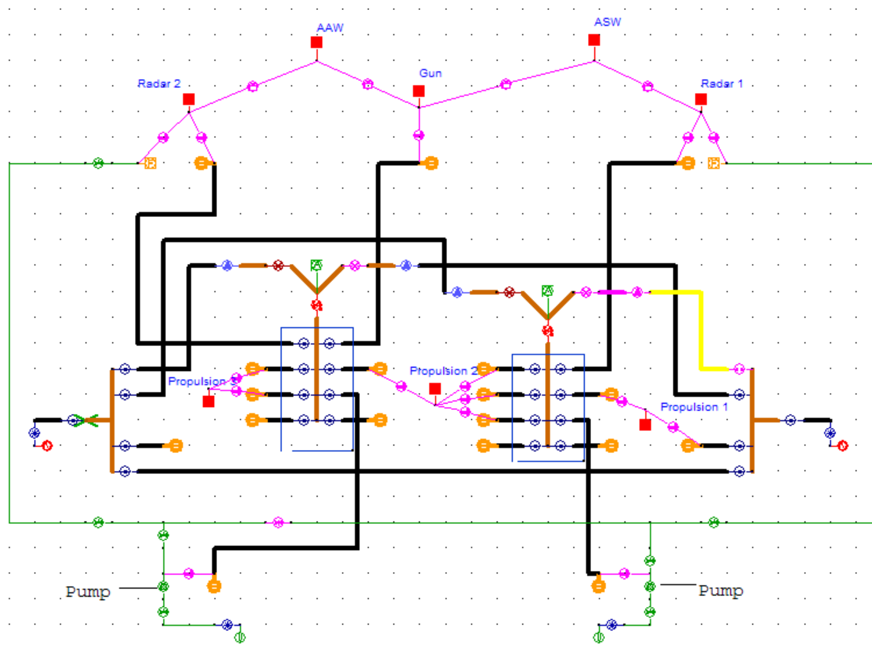

Figure 9: Second Test System – Electrical and Fluid, With Dependencies and Missions

5.4 Example 4: Integrated Model With Loops

The model displayed in Figure 11 is that of a simple integrated model, with a fluid load dependant on an electrical load. In this system, the electrical load has a requirement of 60,000kW distributed evenly across each phase. However, the impedances set on the lines are set such that neither electrical source is able to provide that much power to the load on its own, nor are they able to when working in concert and both switches are turned on. When one switch is turned on, the load is able to draw 46,482kW, or 77.47% of demand. When both are turned on, the load is able to draw 49,407kW, or 82.35% of demand.

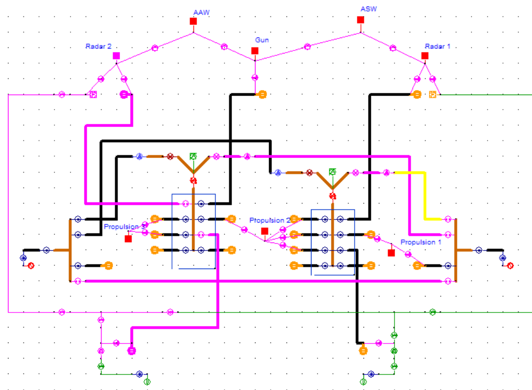

Figure 10a: From Loads Result

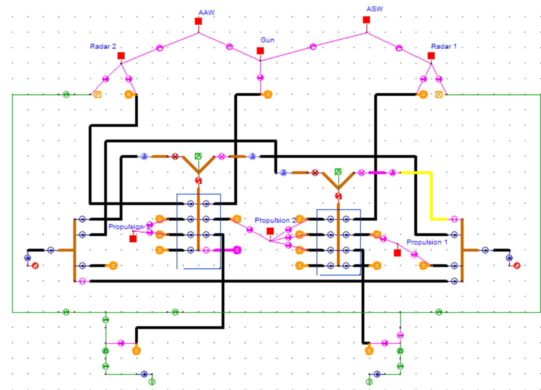

Figure 10c: Hybrid Result

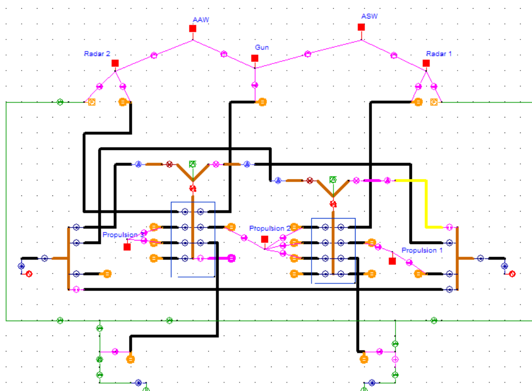

Figure 10b: From Sources Result

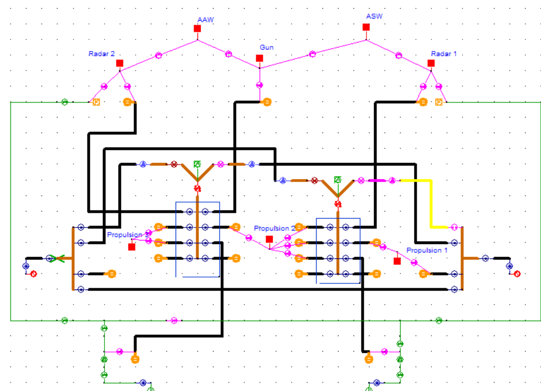

Figure 10d: Cotree Switch Result

Figure 10: Algorithm Results for Example 2

The purpose of this example is to demonstrate the ability of the AddLoops part of the reconfiguration algorithms described in this dissertation to restore loads which can't be restored from only a single source, and to demonstrate its ability to respect partial dependencies: where a component depends on a supporting load, but only requires that load to have a certain percentage of its demand in order to function.

Because of the inability of the system to fully supply the electrical load, if the fluid load requires the electrical load to be fully satisfied (p→AD→first→percent==100) when any reconfiguration algorithm is run on it, they all result in both loads being provided service with the electrical load receiving service through just one switch. However, despite configuring the system such that service is provided to the

loads, they are not considered restored (p→status==ON) because the electrical load is underfed and the fluid load's dependency is unsatisfied (Fig. 12a).

If the fluid load requires the electrical load to only receive 75% of its demand to satisfy the dependency (p→AD→first→percent==75), all algorithms result in the system being configured in the same way as when the fluid load required the electrical load be 100% serviced. However in this case

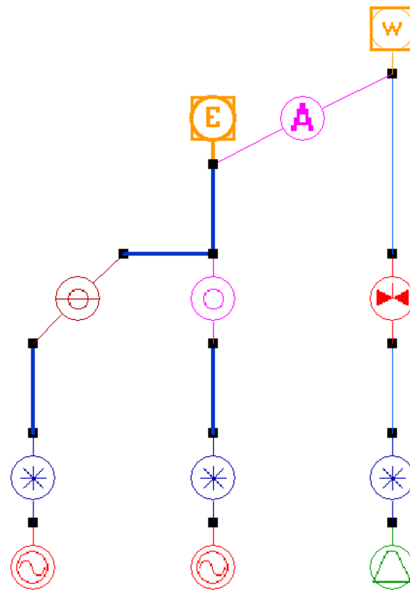

Figure 11: Example 3 System

they recognize both loads as being restored (Fig. 12b) since the electrical load is sufficiently serviced for the purposes of this system: that of satisfying the fluid load's dependency.

When the service requirement the fluid load places on the electrical load is raised to 80% (p→AD→first→percent==80), radial configuration of the system is no longer enough. If reconfiguration wishes to restore the fluid load in this case, the AddLoops algorithm must turn on both switches in the electrical system in order to fully restore the system, which it does (Fig. 12) for all four algorithms.

Reconfiguration Statistics
Total Loads: 2
Total Operable Sectionalizing Devices: 3
Loads Initially Dropped By Failures: 0
Total Loads Serviced: 0

Figure 12a: Ex. 3 Reconfiguration Results, 100% Dependency Requirement

Reconfiguration Statistics
Total Loads: 2
Total Operable Sectionalizing Devices: 3
Loads Initially Dropped By Failures: 0
Total Loads Serviced: 2

Figure 12b: Ex. 3 Reconfiguration Results, 75% Dependency Requirement

Figure 12: Example 3 Results at 100% and 75% Requirements

Reconfiguration Statistics
Total Loads: 2
Total Operable Sectionalizing Devices: 3
Loads Initially Dropped By Failures: 0
Total Loads Serviced: 2

Figure 13: Ex. 3 Reconfiguration Results, 80% Dependency Requirement

5.5 Complexity

The number of operations necessary for each of the algorithms presented in this dissertation to complete varies depending on a number of inputs. The number of loads, switches, interdependencies, adjacencies, and the total number of components each can have a significant impact on execution. However, due to the different approaches taken by each algorithm, the impact of each of these factors can vary. In this section, the following notation is used:

n = # of components
o = # of loads
c = # of adjacencies
s = # of switches
d = # of interdependencies

Table 15 summarizes the complexity of the various functions called by the algorithms.

From Loads Complexity Analysis

The complexity of the From Loads algorithm is primarily driven by the number of components in the system, but other factors can significantly affect it. While several of the functions it uses are fairly simple and operate in O(n) (such as Segment, FailureInFedSegment, ClearPriorities, and CreatePaths),

Function	Complexity	Function Name	Complexity
From Loads	$ocdn^6 + odsn^4 + ond*\log(o)$	Backup	n+s
From Sources	$od^2n^2 + so + s^3n^2 + s^2n^3d^2$	RestoreDependencies	d*O(RestoreCmp)
Hybrid	$o*\log(o)+s^2n^3+son^4d^2+ocdn^7+ocn^8$	ProcessDecPt	n+s+d*O(RestoreCmp)
Cotree Switch	$ond*\log(o) + son^2d^2 + s^3n + cs*\log(s) + csn$	ProcessPath	$n^2+ns+dn*O(RestoreCmp)$
Add Loops	$ond*\log(o) + on^3 + ons*\log(s) + on^2sd$	RestoreCmp	$cdn^6 + dsn^4$
Segment	n	Process Sources	$s^2n^2 + son^3d^2$
FailureInFedSegment	n	ValidBoundSwitches	$snd + s*\log(s)$
ClearPriorities	n	UnlockSwitches(int)	$ocdn^5 + ocn^6$
PropagatePriorities	n^2d^2	UnlockSwitches (Component, set of Component)	$cdn^5 + cn^6$
ChoosePathStart	cn^3	FindOpSwt	s^2n
AreConnectedNoFailures	cn^2	IsUnderfed	nd
CreatePaths	n		

Table 15: Complexity of Reconfiguration Functions

others are more complicated or are affected by more than just the number of components. One such more complex function used by all the algorithms is PropagatePriorities, which operates at $O(n^2d^2)$.

AreConnectedNoFailures is a recursive function which must perform n^2 operations in the worst case in each recursion, but in the worst case must also recur c times, resulting in an $O(cn^2)$. Since it is called n times by the ChoosePathStart function, ChoosePathStart has a worst case complexity of $O(cn^3)$. The Backup function, on the other hand, has a complexity of O(n+s) because it performs a pair of linear operations on all components, followed by a linear operation on switches specifically.

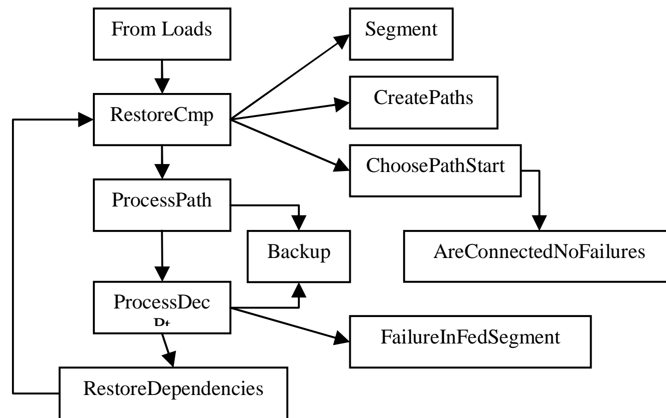

Figure 14:From Loads Function Heirarchy

The remaining functions, RestoreCmp, ProcessPath, ProcessDecPt, and RestoreDependencies are part of a recursion chain as shown in Figure 14. RestoreCmp calls, among others, ProcessPath. ProcessPath calls ProcessDecPt, and ProcessDecPt calls RestoreDependencies which calls RestoreCmp again. RestoreDependencies has a complexity of O(d*O(RestoreCmp)) due to needing to call RestoreCmp on components across interdependencies. ProcessDecPt makes calls to FailureInFedSegment, RestoreDependencies, and Backup, which gives it a complexity of O(n+s+d*O(RestoreCmp)).

ProcessPath must perform a operation of O(n) and make a call to ProcessDecPt and Backup all a total of n times, giving it a complexity of $O(n^2+ns+dn*O(RestoreCmp))$. Lastly, RestoreCmp must call ChoosePathStart and ProcessDecPt n times in the worst case for a complexity of $O(cn^4+n^2s+dn^2*O(RestoreCmp))$, which reduces to $O(cdn^6 + dsn^4)$.

RestoreCmp is called by the main From Loads algorithm once for each load, giving that operation a complexity of $O(ocdn^6 + odsn^4)$. The only other operation the From Loads algorithm performs that is more than O(n) is the initial sorting of loads at O(ond*log(o)), so the overall complexity of From Loads is $O(ocdn^6 + odsn^4 + ond*log(o))$.

From Sources Complexity Analysis

The complexity of the From Sources algorithm is most heavily influenced by the number of components in the system, but other subsidiary factors can also have an effect. Like From Loads, From Sources makes use of the simple, O(n) functions Segment and ClearPriorities. It also uses two other, more complex functions in ProcessSources and ValidBoundSwitches.

ValidBoundSwitches is a function which, for a given source, determines which of the switches bounding its service area can be used by reconfiguration to try and expand that service area (discouting potential flow constraint violations). This requires a series of linear operations on the number of switches (s) fed by the source, but also requires checking components (n) in the segment potentially fed

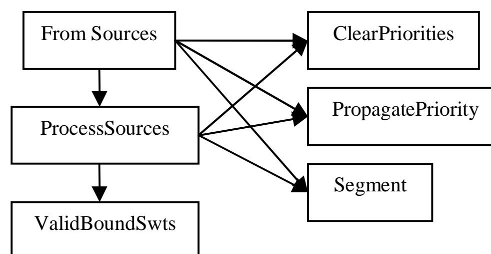

Figure 15:From Sources Function Heirarchy

by a switch in order to determine that their interdependencies (d) are satisfied. The remaining switches must then be sorted according to their priority (s*log(s)). This results in a total complexity for ValidBoundSwitches of O(snd + s*log(s)).

ProcessSources is the core of the From Sources algorithm. For each source in the system (n), it must call ValidBoundSwitches to determine potential candidates for operation. For each of these candidates (s), ProcessSources then perform a series of checks that are linear on the number of components in the system (n), an operation of order sn, and in the worst case must also call PropagatePriorities at $O(n^2d^2)$. The result of this is that ProcessSources has an overall complexity of $O(n*(O(ValidBoundSwitches) + s(n + sn + n^2d^2)))$, which collapses to $O(s^2n^2 + son^3d^2)$.

From Sources mostly contains operations which are linear on the number of components in the system (n). The function call heirarchy for From Sources is shown in Figure 15. It calls PropagatePriorities once for each load ($O(on^2d^2)$), and has an n^3 operation when it initializes the boundary switches for the sources. However, the complexity of these operations are eclipsed by the main loop which iterates a maximum s times (once for each switch in the system). This loop performs a sort on the sources (O(n*log(n))), and then makes a call to ProcessSources followed by some linear operations on the number of components (n) and number of loads (o). This loop has an overall complexity of $O(so + s^3n^2 + s^2n^3d^2)$, which makes the final worst-case complexity of the From Sources algorithm $O(od^2n^2 + so + s^3n^2 + s^2n^3d^2)$.

Hybrid Complexity Analysis

Because the Hybrid algorithm combines aspects of both From Loads and From Sources, it is influenced by a number of factors, but like the previous two algorithms the greatest influence on its complexity is the number of components. Hybrid makes use of several functions previous described, including PropagatePriorites at $O(on^2d^2)$, AreConnectedNoFailures at $O(cn^2)$, ProcessSources at $O(s^2n^3 + osn^3)$, and CreatePaths, ClearPriorities, Segment, and FailureInFedSegment at O(n). Hybrid also makes use of a ValidBoundSwitches function like the one used by From Sources which only differs from that version by a constant operation, and so still has a complexity of O(snd+s*log(s)). Hybrid's function call heirarchy is shown in Figure 16.

The new functions used by Hybrid are UnlockSwitches(int) and UnlockSwitches(Component, set of Component). The former unlocks all switches that could feed loads of a given or higher priority, and iteratively calls the latter on each load with a sufficiently high priority to unlock switches which could feed that load. Thus, UnlockSwitches(int) has a complexity of O(o*O(UnlockSwitches(Component, set of Component))). UnlockSwitches(Component, set of Component) starts by calling CreatePaths on the

load it is given (O(n)), then for each of those potential paths, calls AreConnectedNoFailures to generate the full path and UnlockSwitches(Component, set of Component) on any components on which the components on the path are dependant, as well as on one component in each segment along that path. This gives UnlockSwitches(Component, set of Component) a complexity of $O(cn^3 + (dn^2 + n^3)$*UnlockSwitches(Component, set of Component)), which collapses to $O(cdn^5 + cn^6)$. This gives UnlockSwitches(int) a complexity of $O(ocdn^5 + ocn^6)$.

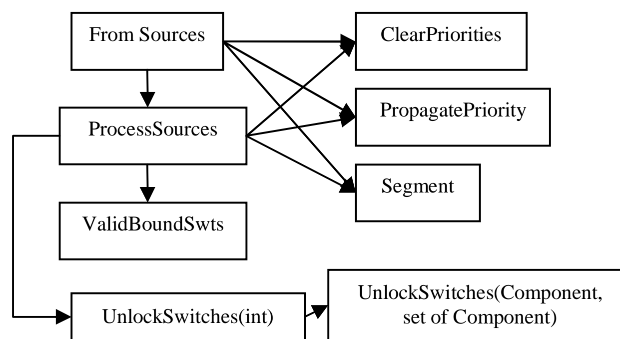

Figure 16:Hybrid Reconfiguration Heirarchy

The main Hybrid function performs a number of operations, but the only ones that are non-dominated in terms of complexity are the sorting of loads at O(o*log(o)), and the main processing loop. The main processing loop is dominated by a call to ProcessSources and UnlockSwitches(int), and must be executed order n times, which gives it a complexity of $O(n(s^2n^2 + son^3d^2 + ocdn^6 + ocn^7))$. Consequently, the worst-case complexity of the Hybrid algorithm is $O(o*\log(o)+s^2n^3+son^4d^2+ocdn^7+ocn^8)$.

Cotree Switch Complexity Analysis

Unlike the previous three algorithms which are dominated by the number of components in the system, Cotree Switch is influenced most strongly by the number of switches specifically. Cotree Switch again starts off with some operations which have been seen before, including sorting the loads by priority at O(ond*log(o)), ClearPriorities at O(n), and calling PropagatePriorities on each load at on^2d^2.

The new function used by Cotree Switch is FindOpSwt, which is used to determine which switch to operate to alleviate a violation. FindOpSwt contains a number of order n and s operations, but is dominated by an order s^2n operation where it eliminates any candidate switches that are in the feeder path of another candidate switch. Thus, FindOpSwt has a complexity of $O(s^2n)$.

There are two main loops in Cotree Switch. The first is executed order s times, and contains a sorting of components, a call to FindOpSwt, a call to ClearPriorities, and calling PropagatePriorities once for each load for a total complexity of $O(sn*log(n) + s^3n + sn + son^2d^2)$. The other loop is executed order c times, and reduces to a sort on switches (O(s*log(s))) and an order n operation order s times for a total complexity of O(cs*log(s) + csn). Thus, Cotree Switch has an overall complexity of $O(ond*log(o) + son^2d^2 + s^3n + cs*log(s) + csn)$.

Add Loops Complexity Analysis

Add Loops makes extensive use of Segment and a new function called IsUnderfed. IsUnderfed which checks the flow level of a component and those components supporting it for a complexity of O(nd).

Add Loops itself first calls IsUnderfed on order n components, then sorts order o components as seen in the load sorting in From Loads at O(ond*log(o)). This is followed by a loop which is executed order o times on the underfed loads. In this loop in the worst case, Add Loops calls IsUnderfed and also checks that the load does not have a failure in its segment in an order n^2 operation. The loop then has another embedded loop that executes order n times, each time taking in the worst case order n^2 + s*log(s) + snd operations. This gives Add Loops a total worst case complexity of $O(ond*log(o) + on^3 + ons*log(s) + on^2sd)$.

Effect of Interdependencies on Algorithm Complexity

Dependencies have a substantial impact on the worst-case complexity of these algorithms. PropagatePriorities, for example, is normally an $O(n^2d^2)$ function which is very commonly called. However, if there are no interdependencies in the system, it collapses to a simple O(n) function because it no longer needs to propagate down supporting components as well as the original component's feeder path. Many instances of recursion and many of the most complex statements involve performing some action, and then performing that same action or another related action on supporting components on the other side of an interdependency. RestoreCmp, for example, doesn't recur at all if there are no interdependencies in the system.

The overall effect of cutting out interdependencies is to significantly reduce overall complexity of the algorithms. From Loads drops to $O(on*log(o) + ocn^3 + osn^2)$, From Sources drops to $O(s^2n^3 + osn^3)$, Hybrid to $O(o*log(o) + s^2n^3 + osn^3 + ocn^7)$, Cotree Switch to $O(n^2 + sn*log(n) + s^3n + son + cs*log(s) + csn)$, and Add Loops to $O(on*log(o) + on^3 + ons*log(s) + osn^2)$. The biggest beneficiaries of dropping interdependencies from the system in terms of worst-case complexity are From Loads, which loses an n^6 term, and Hybrid, which loses an n^8 term.

5.6 Algorithm Comparison vs. Complexity Factors

Analysis of the algorithms presented in this paper indicates that they are at worst polynomial time in the average case. Figures 17 through 21 show the results of execution in terms of runtime of each algorithm on systems of varying sizes with respect to the number of loads, switches, total components, interdependencies, and adjacencies in those systems (overall # of components is held constant for dependency and adjacency analysis). The curves in the figures are best fit trendlines generated by Microsoft Excel. The From Loads, From Sources, and Cotree Switch algorithms in particular

demonstrate a strong polynomial relationship, while the Hybrid algorithm is relatively weak in its trendline, but still no more than polynomial.

Theoretical analysis indicates that the complexity of each algorithm is quadratic with respect to the number of interdependencies and linear with respect to the number of adjacencies. However, the algorithms are not equally affected by the number of each present, as for example every term of the From Loads complexity function is affected by interdependencies, compare to only two for From Sources. Figures 20 and 21 show the response of these algorithms to a system with around 50,000 components as dependencies or adjacencies were randomly added to it, and shows that From Loads and Cotree Switch were the algorithms most heavily influenced by the presence of interdependencies, while Cotree Switch and Hybrid were the only algorithms significantly affected by the number of adjacencies. The trendline for the Cotree Switch algorithm on Figure 20 ignores the two outliers.

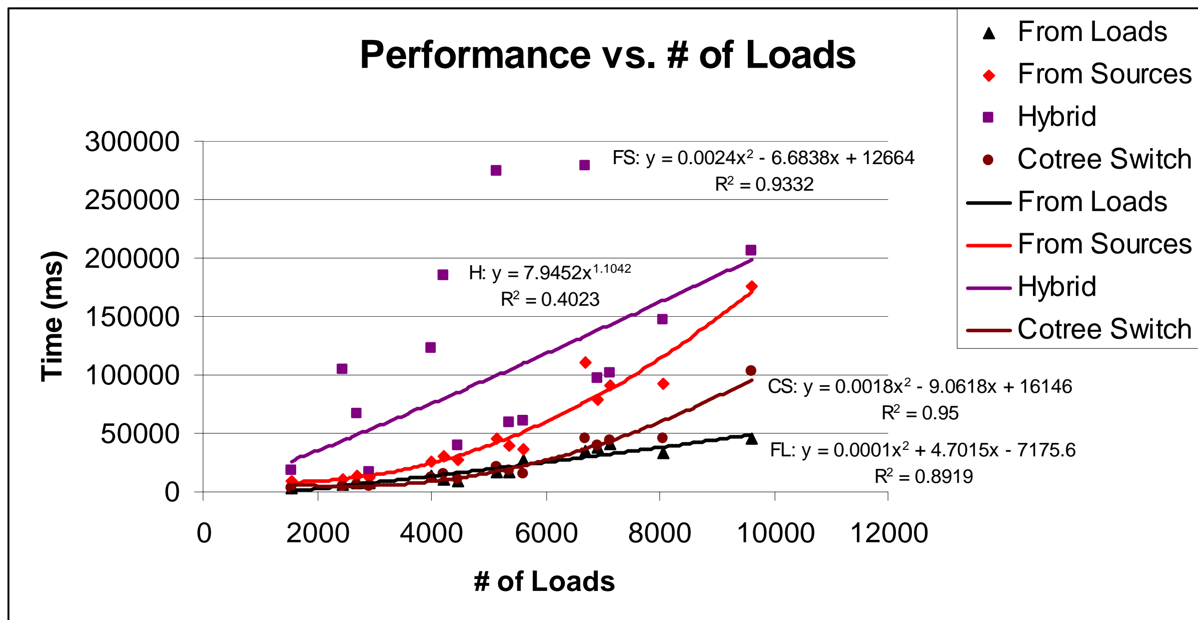

Figure 17: Algorithm Performance versus Number of Loads

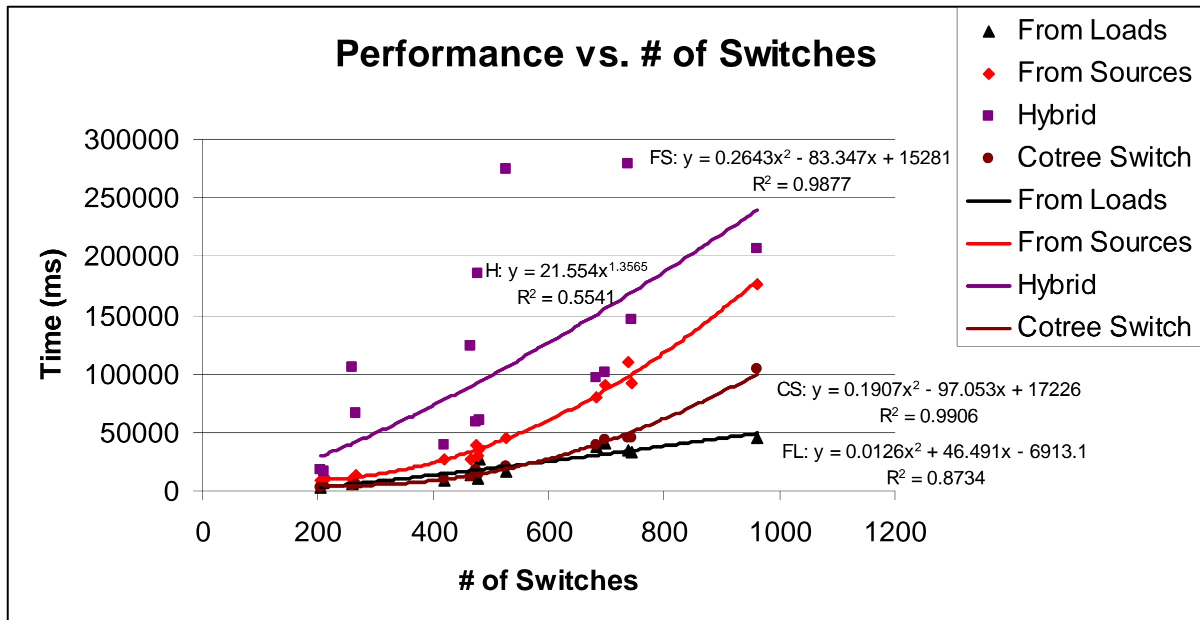

Figure 18: Algorithm Performance versus Number of Switches

The reasons for deviation of the empirical results in these cases from theoretical analysis are primarily threefold. The theoretical analysis provides for algorithm complexity in a worst-case scenario, which would involve every component having an adjacency or every component being dependent on every other component. Such systems simply do not exist. The worst-case analysis also assumes things such as no restoration path in From Loads is valid except the last one examined, which boosts the complexity over what would actually be seen. In addition, the presence of interdependencies and adjacencies can limit the possible options the algorithms can explore, which can significantly decrease the number operations necessary to reach a solution.

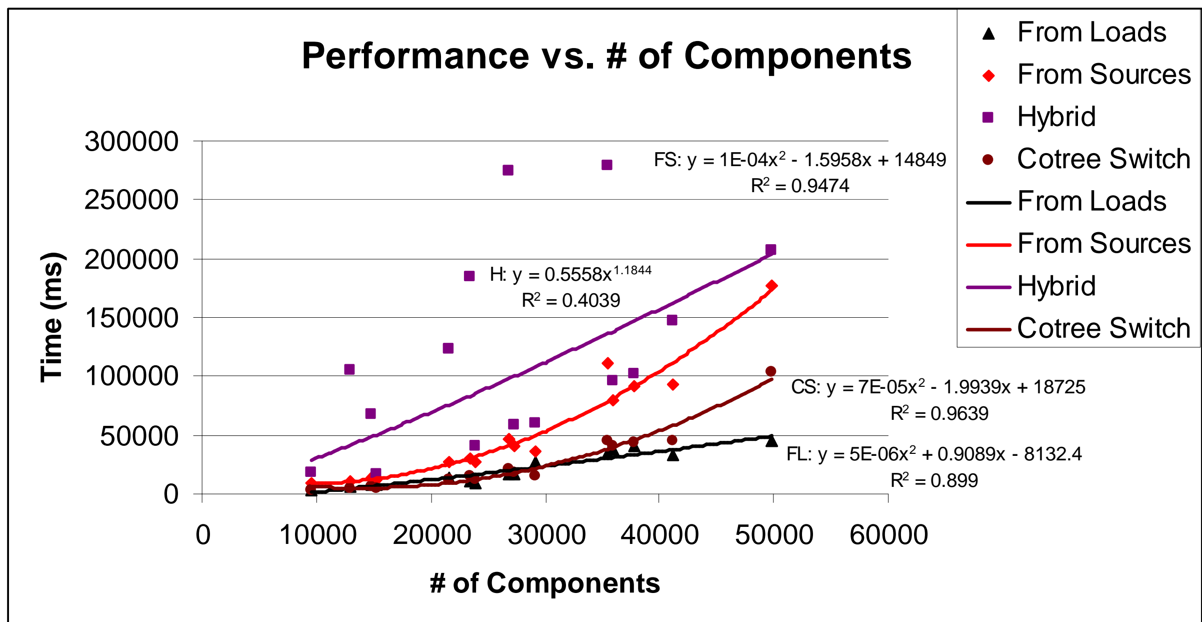

Figure 19: Algorithm Performance versus Number of Components

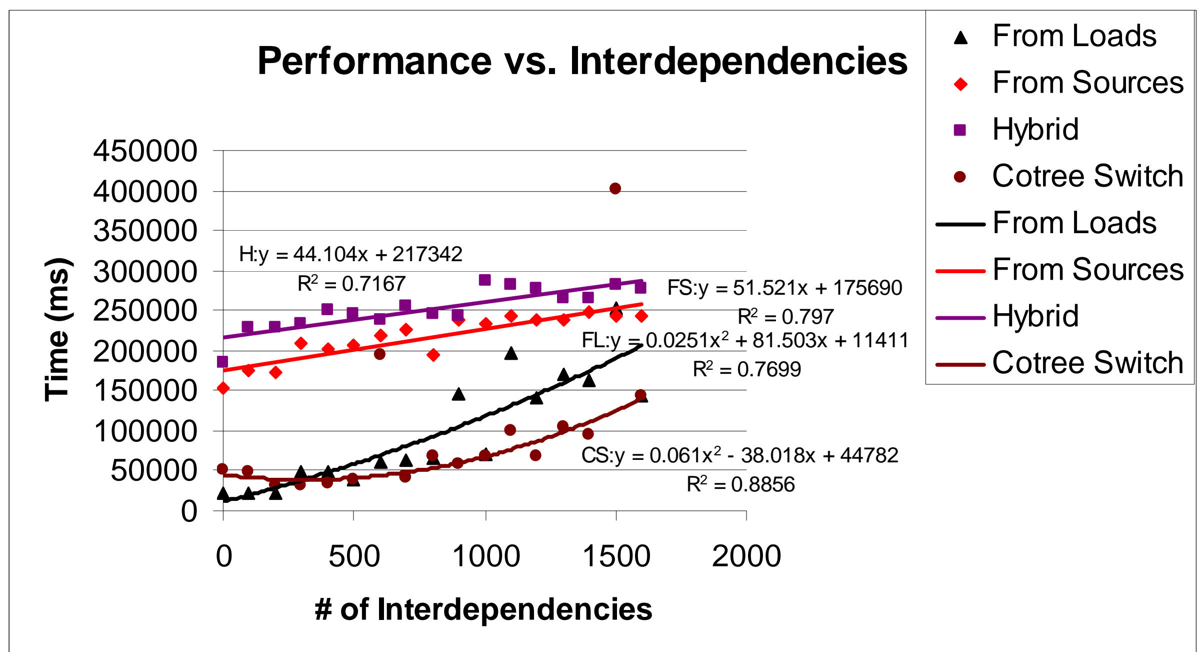

Figure 20:Algorithm Performance versus Number of Interdependencies

The difficulty of dealing with interdependencies is particularly evident in each algorithm. Even though From Sources and Hybrid seem to be significantly less affected by interdepenencies than From

Loads or Cotree Switch, they still see large time increases of up to 50% as the number of interdependencies is increased.

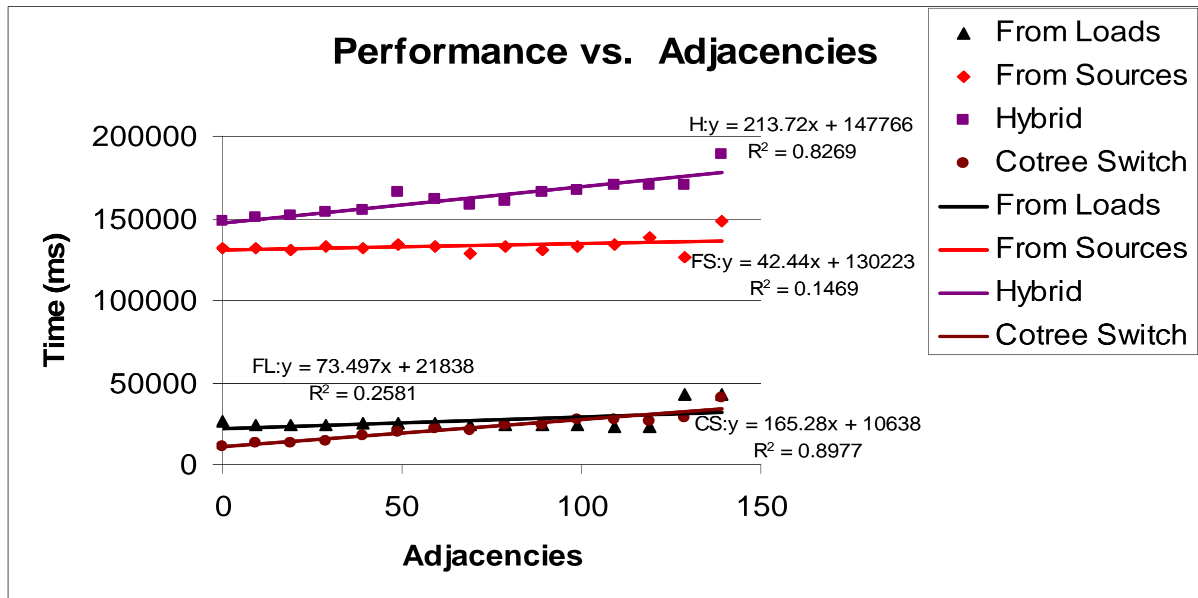

Figure 21:Algorithm Performance versus Number of Adjacencies

Figure 22 shows the degradation of the performance of the algorithms as interdependencies are increased. Each algorithm restores all or nearly all possible restorable loads up until a point, at which the number of restored loads drops off precipitously before levelling out. For From Loads, From Sources, and Hybrid, this breakpoint occurs after the first few hundred interdependencies, while the Cotree Switch algorithm lasts significantly longer before succumbing to the complexity the interdependencies add to the system. The reason for this decrease in performance has to do with the way interdependencies affect the order of load restoration. Satisfying interdependencies first can result in some loads being unrestorable which might have been restorable if restoration had been attempted in a different order or with different restoration paths. The reason for the steepness of the dropoff is the increased probability of circular interdependency chains (A depends on B depends on C depends on A). Such chains are difficult for the algorithms to resolve and can lead to unrestored load that could be

restored by an algorithm capable of finding an optimal solution. Fortunately, the likelyhood of a system of this size having that many interdependencies is very low.

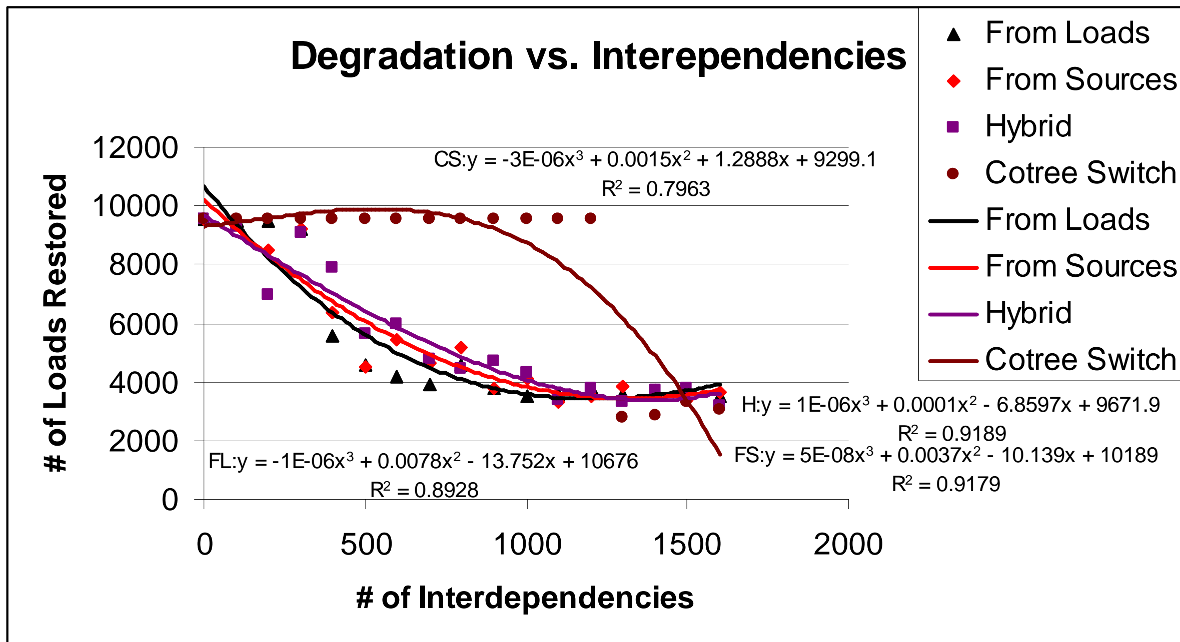

Figure 22:Algorithm Performance Degradation vs. Interdependencies

The reason for the difference in the dropoff point between Cotree Switch and the other algorithms has to do with the way they handle restoration paths. The Cotree Switch algorithm starts from a system with all switches turned on, and then turns off switches as long as doing so causes no constraint violations or additional dropped loads. By contrast, From Loads, From Sources, and Hybrid reach a restoration solution by turning on switches in order to restore load. In other words, Cotree Switch starts with all paths restored and tries to prune unnecessary ones while the other algorithms seek to build a set of restoration paths. While the former is not guaranteed to result in a radial solution (a desireable condition for many utilities), it does create less potential for restoring loads in an order which prevents otherwise restorable loads from being restored.

5.7 Performance Observations

The first two examples examining the models shown in Figures 7 and 8 reveal some valuable information about the algorithms. First, each performs better than the DAOP method in almost every examined metric (the sole exception being phase balancing, where DAOP is in the middle of them). In the first example, all four proposed algorithms are substantially better than DAOP at minimizing losses – the very thing DAOP was designed to do [19,20]. Furthermore, they all feature lower losses than the DAOP solution while providing substantially more power to the system due to the fact that they restore more load.

Of the four proposed algorithms, the best performers are From Sources and Cotree Switch. The Cotree Switch algorithm is in general faster than the others by a substantial margin (except for the two stage From Loads solution for example 2), and also performs best in phase imbalance and kW losses. However, the From Sources algorithm demonstrates better load balancing across the sources, and in general produces solutions which require fewer switching operations to implement, a significant advantage in implementing solutions in practice.

The speed advantage possessed by the Cotree Switch algorithm is related to the fact that it starts with a system that has all switches turned on. Instead of checking system constraints every time a load is picked up as switches are turned on as in the other algorithms, Cotree Switch only needs to check constraints each time it drops a load when isolating failure or breaks an independent loop. Because the number of potential loops in a real-world system is limited (only 82 in the large example, which has over 50,000 components), the Cotree Switch algorithm is going to spend significantly less time checking constraints, which is the most expensive part of the algorithms. As an example, the single stage run of the From Sources algorithm on the larger example model spends 97% of its time checking constraints

calling the power flow algorithm over 9000 times, while the Cotree Switch algorithm only calls power flow 245 times, taking up only 77% of the total time.

However, the speed difference of a couple minutes between the algorithms on the larger model may not actually be significant in a real-world implementation, particularly since the algorithms are likely to be run on a much more powerful computer than that used for testing. The few seconds difference on the smaller model is almost certainly insignificant.

The From Loads and Hybrid algorithms lag behind the other, but were designed with highly-prioritized systems in mind, a condition which does not apply to the real-world examples presented here. It is possible that they would show a better performance on a larger system that was more fully prioritized, and in fact the third example on the simple integrated model indicates that this may be the case at least for From Loads.

VI. Conclusion

The problem of how to respond to disruptions in infrastructure systems is an important and common one. Prioritization of loads, interdependencies between systems, and non-linear flows of some systems only complicate it. In fact, as proven in this dissertation in 3.4, the prioritized reconfiguration problem is an NP-hard problem. Many solutions have been proposed for this problem, but no comprehensive solutions have been developed without needing to simplify the system model.

This dissertation has proposed four algorithms for the prioritized reconfiguration of interdependent critical infrastructure systems. These algorithms are designed to use a Graph Trace Analysis model that is an unsimplified representation of the systems being analyzed, and are written using a new notation developed and described in this dissertation specifically for GTA. This notation can be used to write algorithms in GTA that are independent of the systems being analyzed. It is also easy to implement algorithms written this notation using tools such as those provided in the C++ Standard Template Library.

Of all prior works surveyed, very few recognized the need for arbitrary prioritization and only one [4] attempted to address both arbitrary prioritization and system interdependencies. While it was able to achieve results for its example systems very quickly, it was only able to do so through applying many simplifying assumptions to its model, and it was forced to use different models for each of the systems involved.

By contrast, each of the algorithms proposed in this dissertation are capable of quickly solving reconfiguration for large systems without any simplifications being applied to the system model. They also all allow for arbitrary levels of priority on system loads. In addition, the generic nature of GTA allows the proposed algorithms to do so with a single, integrated model containing all systems being examined, complete with the dependencies between them.

The proposed algorithms are further capable of handling dependencies which only need to be partially satisfied as shown in 5.4. In the event that a supporting load can only receive part of the service it normally demands, the proposed algorithms are capable of determining that it should still be provided service if it can be provided enough to satisfy the loads it supports. This recognition and handling of partial dependencies is something which no prior solution has addressed.

Performance testing of these algorithms shows that they are competetive with or better than previous solutions in mutiple metrics (line losses, switching operations, loads restored, etc.), while addressing aspects of the reconfiguration problem (such as arbitrary prioritization and interdependencies, including partial dependency fulfillment) which previous solutions have not. The complexity of the algorithms is demonstrated to be polynomial, with the ability to handle loops and dependencies not causing an unreasonable increase in complexity for realistic systems.

In addition, the proposed algorithms are not redundant, as they have advantages over each other depending on the nature of the model being analyzed. As discussed in 5.5, while the Cotree Switch and From Sources algorithms seem to perform the best in the real-world models tested, they each perform better than each other in different metrics. There is also evidence to suggest that the From Loads and Hybrid algorithms may have advantages over others when considering highly prioritized systems.

6.1 Future Work

Future work with the algorithms proposed in this dissertation will require field experimentation with actual utility systems to see how they deal with actual disruptions. Different kinds of utilities in the same geographic area could be encouraged to work together to develop an integrated systems model complete with dependencies between system types in order to perform fully robust experiments with the proposed algorithms. Furthermore, a standard programming library akin to the C++ Standard Template

Library could be developed to implement the GTA notation proposed in this dissertation in order to ease the writing of further analysis algorithms for GTA models.

VII. References

[1] K.L. Butler-Purry, N.D.R. Sarma and I.V. Hicks, Service Restoration in Naval Shipboard Power Systems, *IEE Proc. Generation, Transmission and Distribution, Vol. 151, Iss. 1*, Jan. 2004, pp. 95-102.

[2] S. Ćurcić, C.S. Özveren and K.L. Lo, Computer-Based Strategy for the Restoration Problem in Electric Power Distribution Systems, *IEE Proc. Generation, Transmission and Distribution, Vol. 144, Iss. 5*, Sep. 1997, pp. 389-398.

[3] S. Khushalani, J. Solanki and N. Shulz, Optimized Restoration of Combined AC/DC Shipboard Power Systems Including Distributed Generation and Islanding Techniques, *Electric Power Systems Research, 78* (2008) 1528-1536.

[4] E.E. Lee, J.E. Mitchell and W.A. Wallace, Restoration of Services in Interdependent Infrastructure Systems: A Network Flows Approach, *IEEE Trans. Systems, Man, and Cybernetics—Part C: Applications and Reviews, Vol. 37, No. 6*, Nov. 2007, pp. 1303-1317.

[5] S.M. Rinaldi, J.P. Peerenboom and T.K. Kelly, Identifying, Understanding, and Analyzing Critical Infrastructure Interdependencies, *IEEE Control Systems Magazine*, Dec. 2001, pp. 11-25.

[6] A. Merlin and H. Back, Search for a Minimal-Loss Operating Spanning Tree Congifuration in an Urban Power Distribution System, *Proc. 5th Power System Computation Conf. (PSCC), Cambridge, UK,* 1975, pp. 1-18.

[7] D. Shirmohammadi and H.W. Hong, Reconfiguration of Electric Distribution Networks for Resistive Line Loss Reduction, *IEEE Trans. Power Delivery, 4* (1989) 1492-1498.

[8] S.K. Goswami and S.K. Basu, A New Algorithm for the Reconfiguration of Distribution Feeders for Loss Minimization, *IEEE Trans. Power Delivery, 7* (1992) 1484-1491.

[9] C.T. Huddleston, R.P. Broadwater and A. Chandrasekaran, Reconfiguration Algorithm for Minimizing Losses in Radial Electric Distribution Systems, *Electric Power Systems Research, 18* (1990) 57-66.

[10] R.P. Broadwater, A.H. Khan, H.E. Shaalan and R.E. Lee, Time Varying Load Analysis to Reduce Distribution Losses Through Reconfiguration, *IEEE Trans. Power Delivery, 8* (1993) 294-300.

[11] A. Augugliaro, L. Dusonchet and S. Mangione, An Efficient Greedy Approach for Minimum Loss Reconfiguration Distribution Networks, *Electric Power Systems Research, 35* (1995) 167-176.

[12] J.A. Martín and A.J. Gil, A New Heuristic Approach for Distribution Systems Loss Reduction, *Eletric Power Systems Research, 78* (2008) 1953-1958.

[13] S. Civanlar, J.J. Grainger, H. Yin and S.S.H. Lee, Distribution Feeder Reconfiguration for Loss Reduction, *IEEE Trans. Power Delivery, 3* (1988) 1217-1223.

[14] T.E. Lee, M.Y. Cho and C.S. Chen, Distribution System Reconfiguration to Reduce Resistive Losses, *Electric Power Systems Research, 30* (1994) 25-33.

[15] C.A. Castro and A.A. Watanabe, An Efficient Reconfiguration Algorithm for Loss Reduction of Distribution Systems, *Electric Power Systems Research, 19* (1990) 137-144.

[16] K. Kim, Y. Ko and K.H. Hung, Artificial Neural Network Based Feeder Reconfiguration for Loss Reduction in Distribution Systems, *IEEE Trans. Power Delivery, 8* (1993) 1356-1366.

[17] K. Nara, T. Satoh and M. Kitagawa, Distribution System Loss Minimum Re-configuration by Genetic Algorithm, *Proc. 3oh Symp. Expert Systems Application to Power Systems (ESAPS), Tokyo and Kobe, Japan, 1991*, pp. 724-730.

[18] T. Taylor and D. Pubkeman, Implementation of Heuristic Search Strategies for Distribution Feeder Reconfiguration, *IEEE Trans. Power Delivery, 5* (1990) 239-246.

[19] W.M. Lin and H.C. Chin, A New Approach for Distribution Feeder Reconfiguration for Loss Reduction and Service Restoration, *IEEE Trans. Power Delivery, Vol. 13, No. 3,* July 1998, pp. 870-875.

[20] T.E. McDermott, I. Drezga and R.P. Broadwater, A Heuristic Nonlinear Constructive Method for Distribution System Reconfiguration, *IEEE Trans. Power Delivery, Vol. 14, Iss. 2,* May 1999, pp. 478-483.

[21] R.P. Broadwater, P.A. Dolloff, T.L. Herdman, R. Karamikhova and A.F. Sargent, Minimum Loss Optimization in Distribution Systems: Discrete Ascent Optimal Programming, *Electric Power Systems Research, 36* (1996) 113-121.

[22] R. Sampath, H. Darabi, U. Buy, and J. Liu, Control Reconfiguration of Discrete Event Systems With Dynamic Control Specifications, *IEEE Trans. Automation Science and Engineeing, Vol. 5, No. 1,* Jan. 2008, pp. 84-100.

[23] A. Augugliaro, L. Dusonchet, E. Riva Sanseverino, Service Restoration in Compensated Distribution Networks Using a Hybrid Genetic Algorithm, *Electric Power Systems Research, 46* (1998) 59-66.

[24] J.Z. Zhu, Optimal Reconfiguration of Electrical Distribution Network Using the Refined Genetic Algorithm, *Electric Power Systems Research, 62* (2002) 37-42.

[25] Y. Kumar, B. Das and J. Sharma, Service Restoration in Distribution System Using Non-dominated Sorting Genetic Algorithm, *Electric Power Systems Research, 76* (2006) 768-777.

[26] H.C. Chang and C.C. Kuo, Network Reconfiguration in Distribution Systems Using Simulated Annealing, *Electric Power Systems Research, 29* (1994) 227-238.

[27] C.T. Su and C.S. Lee, Feeder Reconfiguration and Capacitor Setting for Loss Reduction of Distribution Systems, *Electric Power Systems Research, 58* (2001) 97-102.

[28] D. Zhang, Z. Fu and L. Zhang, An Improved TS Algorithm for Loss-Minimization Reconfiguration in Large-Scale Distribution Systems, *Electric Power Systems Research, 77* (2007) 685-694.

[29] D.J. Shin, J.O. Kim, T.K. Kim, J.B. Choo and C. Singh, Optimal Service Restoration and Reconfiguration of Network Using Genetic-Tabu Algorithm, *Electric Power Systems Research, 71* (2004) 145-152.

[30] C.T. Su, C.F. Chang and J.P. Chiou, Distribution Network Reconfiguration for Loss Reduction by Ant Colony Search Algorithm, *Electric Power Systems Research, 75*(2005) 190-199.

[31] E. Carpento and G. Chicco, Distribution System Minimum Loss Reconfiguration in the Hyper-Cube Ant Colony Optimization Framework, *Electric Power Systems Research, 78* (2008) 2037-2045.

[32] K. Huang, S.K. Srivastava, D.A. Cartes and L.H. Sun, Market-Based Multiagent system for Reconfiguration of Shipboard Power Systems, *Electric Power Systems Research, 79* (2009) 550-556.

[33] J. Zhu, X. Xiong, J. Zhang, G. Shen, Q. Xu and Y. Xue, A Rule Based Comprehensive Approach for Reconfiguration of Electrical Distribution Network, *Electric Power Systems Research, 79* (2009) 311-315.

[34] Z. Ding, S.K. Srivastava, D.A. Cartes and S. Suryanarayanan, Dynamic Simulation Based Analysis of a New Load Shedding Scheme for a Notional Destroyer Class Shipboard Power System, *IEEE Electric Ship Technologies Symposium, 2007,* May 2007, pp. 95-102.

[35] L.R. Feinauer, K.J. Russell, and R. Broadwater. Graph trace Analysis and Generic Algorithms for Interdependent Reconfigurable System Design and Control. *Naval Engineers Journal, Vol. 120, Iss. 1,* March 2008.

[36] J.B. Warmer and A.G. Kleppe, The Object Constraint Language: Precise Modeling With UML, Addison-Wesley, Reading, MA, 1999.

[37] M. Sipser, Introduction to the Theory of Computation: Second Edition, Thomson Course Technology, Boston, MA, 2006.

[38] M. Garey and D. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, Freeman, San Francisco, CA, 1979.

Appendix A – Reconfiguration Algorithms

Utility Functions

0. **Segment()**
1. **ClearPriorities()**
2. **PropagatePriority()**
3. **CheckConstraints()**

Segment(Component Cmp)

```
0. SegStart=FPT(Cmp)→collect(p|p→type==SWITCH || FPT(p)→size==0)→first
1. return
       FT(SegStart)→collect(p|FPT(p)→collect(q|q→type==SWITCH)→first=={SegStart})→including({SegStart})
```

ClearPriorities(Model M)

```
0. M→iterate(p| If p→type!=LOAD
                Then p→priority=0
                EndIf)
```

FailureInFedSegment(Component pCmp)

```
0. If Segment(pCmp→pF) →collect(p|p→status==FAILED)
    Then return TRUE
    Else return FALSE
```

PropagatePriority(Component pCmp)

```
0. FPT(pCmp)→iterate(p| If p→priority < pCmp→priority
                        Then p→priority = pCmp→priority
                        EndIf,
                        p→AD→iterate(q| If q→pCmp→priority < p→priority
                                        Then q→pCmp→priority = p→priority,
                                             PropagatePriority(q→pCmp)
                                        EndIf),
                        p→OD→iterate(q| If q→pCmp→priority < p→priority
                                        Then q→pCmp→priority = p→priority,
                                             PropagatePriority(q→pCmp)
                                        EndIf))
```

CheckConstraints(Component pCmp)

CheckConstraints is a function which checks a Component to ensure that it has not violated any constraints as defined by the person implementing Reconfiguration, for purposes of maximizing flexibility and generalization away from specific system types. The only mandatory constraint check is whether a component's flow violates its capacity. CheckConstraints must return a boolean value indicating whether or not pCmp violates any constraints.

From Loads Functions

0. **FromLoads()**
1. **RestoreCmp()**
2. **CreatePaths()**
3. **ChoosePathStart()**
4. **ProcessPath()**
5. **CreateDecPoints()**
6. **ProcessDecPt()**
7. **RestoreDependencies()**
8. **Backup()**
9. **AreConnectedNoFailures()**

From Loads Structures

```
ClsdSwt
      Component pCmp
      int level

DecPt
     Component pCmp
     int failedOrDeps
```

From Loads Globals

0. set of ClsdSwt ClosedSwitches
1. set of Component Restored
2. set of Component BeingRestored

FromLoads(Model M)

```
0. seq of Component Loads = M→collect(p|p→type == LOAD)
1. M→collect(p|p→type == SWITCH)→collect(p| p→status=OFF)
2. Loads→order(p<q if ( p→priority > q→priority OR
                              (M→exists(S→exists(t→AD→exists(u| u→pCmp==q) OR
                                                             t→OD→exists(u| u→pCmp==q)))
                               AND
                               M→forall(S|S→forall(t|t→AD→collect(u| u→pCmp==p)→size==0 AND
                                                                 t→OD→collect(u| u→pCmp==p)→size==0)))))
3. Loads→iterate(ClosedSwitches={}, BeingRestored={}, RestoreCmp(p, 0))
4. AddLoops(M)
```

RestoreCmp(Component Cmp, int level)

```
0. bool backedup=FALSE, restored=FALSE, startingpath=TRUE, pathfail=FALSE;
   Component pCmp=Cmp;
   seq of Component visited, InitPath;
1. If Restored→includes(Cmp)
   Then return TRUE
   EndIf
2. If BeingRestored→includes(Cmp)
   Then return TRUE
   EndIf
3. CmpSeg=Segment(Cmp)
4. If CmpSeg→collect(p|p→status==FAILED)→size > 0
   Then return FALSE
   EndIf
5. CreatePaths(Paths, Cmp)
6. Paths→iterate(p| If ChoosePathStart(pCmp, p, Cmp, startingpath, Paths, InitPath)==FALSE
                    Then return FALSE
                    EndIf,
                    If pCmp!=NULL AND FPT(pCmp)→collect(p|p→status==FAILED)→size>0
                    Then result=ProcessPath(pCmp, Cmp, startingpath, Paths, InitPath, backedup, level, restored)
                    Else startingpath=TRUE
                    EndIf)
7. If restored==FALSE
   Then return FALSE
   Else Restored=Restored→including(Cmp)
          return TRUE
   EndIf
```

CreatePaths(Sequence Paths, Component Cmp)

```
0. Paths = FT(FPT(Cmp)→last)→collect(p|p→adjt!=NULL)
1. Paths→prepend(Cmp→fpt)
2. return TRUE
```

ChoosePathStart(Component pCmp, Component p, Component Cmp, bool startingpath, seq of Component Paths, seq of Component InitPath)

```
0. restored=TRUE, visited={}, InitPath={}
1. If startingpath==TRUE
   Then  If Cmp==p
         Then pCmp=p,
              startingpath=false
         Else pCmp=NULL,
              If AreConnectedNoFailures(Cmp, p, visited, InitPath) == TRUE
              Then pCmp=p→adjt,
                   InitPath=InitPath→reverse,
                   startingPath=FALSE
              Else visited={},
                   InitPath={}
              EndIf
              If pCmp==NULL AND Paths→#index+1==Paths→size
              Then return FALSE
              EndIf
         EndIf
   Else Paths→#index--
   EndIf
2. return TRUE
```

ProcessPath(Component pCmp, Component Cmp, bool startingpath, seq of Component Paths, seq of Component InitPath, bool backedup, int level, bool restored)

```
0. seq of DecPoint DecisionPoints={}
1. DecisionPoints=FPT(pCmp)→collect({p,0}| (p→type==SWITCH AND p→status==OFF)
                                  OR p→OD→size>0 OR p→AD→size>0)→reverse
2. DecisionPoints=DecisionPoints→append(InitPath→collect({p,0}| (p→type==SWITCH AND p→status==OFF) OR
                                                         p→OD→size>0                          OR
                                                         p→AD→size>0)→reverse)
3. DecisionPoints→iterate(p| ProcessDecPt(DecisionPoints, p, pCmp, backedup, startingpath, level, restored)
                      If level==0 AND
                         DecisionPoints→#index+1=DecisionPoints→size AND
                         startingpath==false
                      Then If ClosedSwitches→size>0 AND
                              M→collect(q|CheckConstraints(q)==FALSE)→size>0
                           Then If Backup(pCmp, DecisionPoints, 0)==FALSE
                                Then startingpath=TRUE,
                                     restored=FALSE
                                Else backedup=TRUE
                                EndIf
                           Else Restored = Restored→including(Cmp)
                                return TRUE
                           EndIf
                      Else If #index+1==DecisionPoints→size AND restored==TRUE
                           Then Restored=Restored→including(Cmp)
                                return TRUE
                           EndIf
                      EndIf)
```

```
4.  If DecisionPoints→size==0 AND restored==TRUE
    Then Restored=Restored→including(Cmp)
            return TRUE
    EndIf
```

ProcessDecPt(seq of DecPoint DecisionPoints, Component pCmp, bool backedup, bool startingpath, int level, bool restored)

```
0.  bool mustbackup=FALSE,
    Component curCmp
1.  If backedup==TRUE
    Then DecisionPoints→#index = 0,
            backedup=FALSE
    EndIf
2.  curCmp=DecisionPoints→at(DecisionPoints→#index)→pCmp
3.  If curCmp→type==SWITCH AND curCmp→status==OFF
    Then If curCmp→fpt→status==OFF                                   OR
              (curCmp→adjt!=NULL AND curCmp→adjt→fpt→status==OFF) OR
              curCmp→adjt==NULL
            Then curCmp→status=ON
                    If FailureInFedSegment(curCmp)==TRUE
                    Then mustbackup=TRUE
                    EndIf
                    ClosedSwitches=ClosedSwitches→including({curCmp, level})
            EndIf
    EndIf
4.  If mustbackup==FALSE
    Then mustbackup=RestoreDependencies(DecisionPoints, level, curCmp)
    EndIf
5.  If mustbackup==TRUE
    Then If Backup(pCmp, DecisionPoints, level)==FALSE
            Then startingpath=TRUE,
                    restored==FALSE,
                    DecisionPoints→#index=DecisionPoints→size
            Else backedup=TRUE
            EndIf
    EndIf
```

RestoreDependencies(seq of Component DecisionPoints, int level, Component curCmp)

```
0.  bool mustbackup=FALSE
1.  curCmp→AD→iterate(p| If RestoreCmp(p→pCmp, level+1)==FALSE
                         Then mustbackup=TRUE,
                                 #index=curCmp→AD→size
                         EndIf)
2.  If mustbackup==TRUE
    Then return TRUE
    EndIf
3.  mustbackup=TRUE
4.  curCmp→OD→iterate(p| If RestoreCmp(p→pCmp, level+1)==TRUE
                         Then mustbackup=FALSE,
                                 #index=curCmp→OD→size
                         EndIf)
5.  return mustbackup
```

Backup(Component pCmp, seq of Component DecisionPoints, int level)

```
0.  bool found=FALSE
     int curPt=DecisionPoints→collect(n| DecisionPoints→at(n-1)→pCmp==pCmp)→first
1.  DecisionPoints→riterate(p| If DecisionPoints→#index > curPt
                                Then DecisionPoints→#index=curPt
                                EndIf,
                                If p→pCmp→OD→size > 0 AND p→failedOrDeps+1 < p→pCmp→OD→size
                                Then p→failedOrDeps++,
                                     found=TRUE,
                                     DecisionPoints→#index=0
                                Else p→failedOrDeps=0
                                EndIf)
2.  ClosedSwitches→collect(p| If p→level >= level
                                Then p→pCmp→status=OFF
                                EndIf)
3.  return found
```

AreConnectedNoFailures(Component a, Component b, set of Component visited, seq of Component InitPath)

```
0.  seq of Component Path, set of Component AdjPoints, set of Component Try
1.  If FPT(a)→last==FPT(b)→last
    Then Path = FPT(a)→symmetricDifference(FPT(b))→prepend(a)→append(b)
         If Path→collect(p| p→status==FAILED)→size==0
         Then InitPath = InitPath→append(Path),
              return TRUE
         Else return FALSE
         EndIf
    EndIf
2.  AdjPoints = FT(FPT(a)→last)→collect(p|p→adjt!=NULL)
3.  Try=AdjPoints→excluding(AdjPoints→intersection(visited))
4.  If Try→size==0
    Then return FALSE
    EndIf
5.  visited = visited→union(Try)
6.  Try→iterate(p| Path=FPT(a)→symmetricDifference(FPT(p))→prepend(a)→append(b),
                   If Path→collect(q| q→status==FAILED)→size==0
                   Then If AreConnectedNoFailures(p→adjt, b, visited, Path)==TRUE
                        Then InitPath=InitPath→append(Path),
                             return TRUE
                        EndIf
                   EndIf)
7.  return FALSE
```

From Sources Functions

0. **FromSources()**
1. **ProcessSources()**
2. **ValidBoundSwitches()**

From Loads Structures

```
Src
    Component pCmp
    set of Swt BoundSwts
```

```
Swt
    Component pCmp
    bool visited
```

From Sources Globals

0. **set of Component Restored**
1. **set of Component FailedRestore**
2. **seq of Component Loads**
3. **seq of Src Sources**
4. **int highPrior**

FromSources(Model M)

```
0.  bool actiontaken=TRUE, int oldHP
1.  M→collect(p|p→type == SWITCH)→collect(p| p→status=OFF)
2.  Sources=M→collect({p, {}}| p→type==SOURCE)
3.  Loads=M→collect(p| p→type==LOAD)
4.  Loads→order(p<q if p→priority > q→priority)
5.  Sources=Sources→excluding(p| Segment(p→pCmp)
                  →collect(q|q→status==FAILED)→size>0)
6.  highPrior=Loads→first→priority
7.  ClearPriorities(M)
8.  Loads→iterate(p| PropagatePriority(p))
9.  Sources→iterate(p| p→BoundSwts=M→collect({q,FALSE}| q→type==SWITCH          AND
                                                     q→status==OFF             AND
                                                     ((FPT(q)→collect(r|r→status==ON)→size==
                                                           FPT(q)→size         AND
                                                      FPT(q)→includes(p))      OR
                                                      (q→adjt!=NULL            AND
                                                      FUT(q→adjt)→collect(r|r→status==ON)→size==
                                                           FPT(q→adjt)→size    AND
                                                      FPT(q→adjt)→includes(p))))
10. M→iterate(z| If actiontaken==TRUE
           Then M→#index=0,
                Sources→order(p < q if p→f/p→c < q→f/q→c),
                actiontaken=ProcessSources(),
                If Restored→collect(p|p→priority>=highPrior)→size +
                   FailedRestore→collect(p|p→priority>=highPrior)→size ==
                   Loads→collect(p|p→priority>=highPrior)→size
                Then oldHP=highPrior,
                     If Loads→collect(p|p→priority<highPrior)→size > 0
                     Then highPrior=Loads→collect(p|p→priority<highPrior)→first→priority
                     Else highPrior=0
                     EndIf
                EndIf
           Else M→#index=M→size
           EndIf)
11. AddLoops(M)
```

ProcessSources()

```
0.  bool actiontaken=FALSE, bool found=FALSE, set of Component TrySwts
1.  Sources→iterate(p| TrySwts=ValidBoundSwitches(p),
         TrySwts→iterate(q|
              q→status=ON,
              p→BoundSwts→collect(r|r→pCmp==q)→first→visited=TRUE,
              If Segment(q→ft)→collect(r|r→status==FAILED)→size > 0
              Then q→status=OFF,
              Else If Segment(q→ft)→collect(r|r→type==LOAD)→size > 0
                   Then If M→collect(p|CheckConstraints(p)==FALSE)→size>0
                        Then q→status=OFF,
                             FailedRestore=FailedRestore→append(Segment(q→ft)→collect(r|r→type==LOAD))
                        Else p→BoundSwts=p→BoundSwts→excluding(r|r→pCmp==q),
                             p→BoundSwts=p→BoundSwts→including(
                                       Segment(q→ft)→collect({r,FALSE}|
                                                 r→type==SWITCH AND
                                                 r→status==OFF      AND
                                                 r!=q)),
                             p→BoundSwts=p→BoundSwts→including(
                                       Segment(q→ft)→collect({r→adjt,FALSE}|
                                                 r→adjt!=NULL              AND
                                                 r→adjt→type==SWITCH AND
                                                 r→adjt→status==OFF      AND
                                                 r!=q)),
                             ClearPriorities(M),
                             Loads→iterate(r| PropagatePriority(r)),
                             Restored=Restored→including(Segment(q→ft)→collect(r|r→type==LOAD)),
                             FailedRestore=FailedRestore→excluding(Segment(q→ft)→collect(r|r→type==LOAD)),
                             actiontaken=TRUE,
                             TrySwts→#index=TrySwts→size
                        EndIf
              Else ClearPriorities(M),
                   Loads→iterate(r| PropagatePriority(r)),
                   p→BoundSwts=p→BoundSwts→excluding(r|r→pCmp==q),
                   p→BoundSwts=p→BoundSwts→including(Segment(q→ft)→collect({r,FALSE}|
                                                 r→type==SWITCH AND
                                                 r→status==OFF      AND
                                                 r!=q)),
                   p→BoundSwts=p→BoundSwts→including(Segment(q→ft)→collect({r→adjt,FALSE}|
                                                 r→adjt!=NULL              AND
                                                 r→adjt→type==SWITCH AND
                                                 r→adjt→status==OFF      AND
                                                 r!=q)),
                   actiontaken=TRUE,
                   TrySwts→#index=TrySwts→size
              EndIf))
2.  return actiontaken
```

ValidBoundSwitches(Component pCmp)

```
0.  seq of Component Swts=pCmp→BoundSwts→collect(p→pCmp| p→visited==FALSE AND
                        (p→pCmp→ft==p→pCmp→brt OR p→pCmp→ft→status==OFF) AND
                        (p→pCmp→adjt==NULL OR p→pCmp→adjt→status==OFF)
1.  Swts=Swts→excluding(p| p→adjt!=NULL AND
                          Segment(p→fpt)→collect(q| q→status==FAILED)→size>0) AND
                          Segment(p→adjt)→collect(q| q→status==FAILED)→size>0))
2.  Swts=Swts→excluding(p| p→adjt==NULL AND
          Segment(p→ft)→collect(q|q→AD→collect(r|r→pCmp→status==OFF)→size>0  AND
                         q→priority<highPrior)→size>0                         AND
          Segment(p→ft)→collect(q|q→OD→collect(r|r→pCmp→status==ON)→size==0 AND
                         q→priority<highPrior)→size>0)
3.  Swts=Swts→excluding(p| p→adjt!=NULL AND p→adjt→status==OFF AND
          Segment(p→adjt)→collect(q| q→AD→collect(r|r→pCmp→status==OFF)→size>0
                          AND q→priority<highPrior)→size>0 AND
          Segment(p→adjt)→collect(q| q→OD→collect(r|r→pCmp→status==ON)→size==0
                          AND q→priority<highPrior)→size>0)
4.  Swts=Swts→excluding(p| p→adjt!=NULL AND p→fpt→status==OFF AND
         Segment(p→fpt)→collect(q|q→AD→collect(r|r→pCmp→status==OFF)→size>0  AND
                         q→priority<highPrior)→size>0                         AND
         Segment(p→fpt)→collect(q|q→OD→collect(r|r→pCmp→status==ON)→size==0 AND
                         q→priority<highPrior)→size>0)
5.  return Swts→order(p<q if p→priority>q→priority)
```

Hybrid Functions

0. **Hybrid()**
1. **ProcessSources()**
2. **ValidBoundSwitches()**
3. **UnlockSwitches(int)**
4. **UnlockSwitches(Component, set of Component)**

Hybrid Structures

Src
- Component pCmp
- set of Swt BoundSwts

Swt
- Component pCmp
- bool visited

Hybrid Globals

0. **set of Component Restored**
1. **set of Component FailedRestore**
2. **seq of Component Loads**
3. **seq of Src Sources**
4. **int highPrior**

Hybrid(Model M)

```
0.  bool actiontaken=TRUE, int oldHP
1.  M→collect(p|p→type == SWITCH)→collect(p| p→status=OFF)
2.  Loads=M→collect(p| p→type==LOAD)
3.  Loads→order(p<q if p→priority > q→priority)
4.  Sources=Sources→excluding(p| Segment(p→pCmp)→collect(q|q→status==FAILED)→size>0)
5.  M→collect(p| p→type==SWITCH AND p→operable=FALSE)
6.  highPrior=Loads→first→priority
7.  ClearPriorities(M)
8.  Loads→iterate(p| PropagatePriority(p))
9.  UnlockSwitches(highPrior)
10. Sources=M→collect({p, {}}| p→type==SOURCE)
11. Sources→iterate(p| p→BoundSwts=M→collect({q,FALSE}|
                                   q→type==SWITCH              AND
                                   q→status==OFF               AND
                                   q→operable==TRUE            AND
                                   ((FPT(q)→collect(r|r→status==ON)→size==
                                        FPT(q)→size            AND
                                    FPT(q)→includes(p))        OR
                                   (q→adjt!=NULL               AND
                                    FUT(q→adjt)→collect(r|r→status==ON)→size==
                                        FPT(q→adjt)→size       AND
                                    FPT(q→adjt)→includes(p))))
12. M→iterate(z| If actiontaken==TRUE
            Then M→#index=0,
                Sources→order(p < q if p→f/p→c < q→f/q→c),
                actiontaken=ProcessSources(),
                If Restored→collect(p|p→priority>=highPrior)→size +
                   FailedRestore→collect(p|p→priority>=highPrior)→size ==
                   Loads→collect(p|p→priority>=highPrior)→size
                Then oldHP=highPrior,
                    If Loads→collect(p|p→priority<highPrior)→size > 0
                    Then highPrior=Loads→collect(p|p→priority<highPrior)→first→priority
                    Else highPrior=0
                    EndIf,
                    UnlockSwitches(highPrior),
                    Sources→iterate(p| p→BoundSwts=M→collect({q,FALSE}|
                                   q→type==SWITCH              AND
                                   q→status==OFF               AND
                                   q→operable==TRUE            AND
                                   ((FPT(q)→collect(r|r→status==ON)→size==
                                        FPT(q)→size            AND
                                    FPT(q)→includes(p))        OR
                                   (q→adjt!=NULL               AND
                                    FUT(q→adjt)→collect(r|r→status==ON)→size==
                                        FPT(q→adjt)→size       AND
                                    FPT(q→adjt)→includes(p))))
                EndIf
            Else M→#index=M→size
            EndIf)
13. AddLoops(M)
```

ProcessSources()

As with the From Sources Algorithm

ValidBoundSwitches(Component pCmp)

```
0. seq of Component Swts=pCmp→BoundSwts→collect(p→pCmp| p→visited==FALSE AND
                        (p→pCmp→ft==p→pCmp→brt OR p→pCmp→ft→status==OFF) AND
                        (p→pCmp→adjt==NULL OR p→pCmp→adjt→status==OFF)
1. Swts=Swts→excluding(p| p→operable==FALSE)
2. Swts=Swts→excluding(p| p→adjt!=NULL AND
                        Segment(p→fpt)→collect(q| q→status==FAILED)→size>0) AND
                        Segment(p→adjt)→collect(q| q→status==FAILED)→size>0))
3. Swts=Swts→excluding(p| p→adjt==NULL AND
        Segment(p→ft)→collect(q|q→AD→collect(r|r→pCmp→status==OFF)→size>0  AND
                        q→priority<highPrior)→size>0                         AND
        Segment(p→ft)→collect(q|q→OD→collect(r|r→pCmp→status==ON)→size==0 AND
                        q→priority<highPrior)→size>0)
4. Swts=Swts→excluding(p| p→adjt!=NULL AND p→adjt→status==OFF AND
        Segment(p→adjt)→collect(q| q→AD→collect(r|r→pCmp→status==OFF)→size>0
                        AND q→priority<highPrior)→size>0 AND
        Segment(p→adjt)→collect(q| q→OD→collect(r|r→pCmp→status==ON)→size==0
                        AND q→priority<highPrior)→size>0)
5. Swts=Swts→excluding(p| p→adjt!=NULL AND p→fpt→status==OFF AND
        Segment(p→fpt)→collect(q|q→AD→collect(r|r→pCmp→status==OFF)→size>0  AND
                        q→priority<highPrior)→size>0                         AND
        Segment(p→fpt)→collect(q|q→OD→collect(r|r→pCmp→status==ON)→size==0 AND
                        q→priority<highPrior)→size>0)
6. return Swts→order(p<q if p→priority>q→priority)
```

UnlockSwitches(int priority)

```
0. set of Component visited, seq of Component Paths, seq of Component InitPath,
   seq of Component Feeder, set of Component seen, Component prevSwt, int prevSwtIdx
1. UnlockLoads=Loads→collect(p| p→priority >= priority)
2. UnlockLoads→iterate(p| visited={}, UnlockSwitches(p, visited))
```

UnlockSwitches(Component Cmp, set of Component visited)

```
0. seq of Component Feeder, seq of Component InitPath, set of Component seen, int prevSwtIdx
1. CreatePaths(Paths, p)
2. Paths→iterate(q| seen={}
            If q==p→fpt
            Then InitPath=FPT(q)
            Else If AreConnectedNoFailures(p, q, seen, InitPath)==TRUE
                Then Feeder=FPT(q→adjt)→prepend(q),
                    InitPath=InitPath→append(Feeder),
                EndIf,
                InitPath→iterate(r| If r→status==FAILED
                                    Then InitPath→#index=InitPath→size
                                    Else If r→type==SWITCH
                                        Then prevSwtIdx=InitPath→#index
                                        EndIf
                                    EndIf),
                InitPath→iterate(r| Feeder=Feeder→append(r),
                                    If InitPath→#index=prevSwtIdx
                                    Then InitPath→#index=InitPath→size
                                    EndIf),
```

```
          Feeder→collect(r| r→type==SWITCH AND r→operable=TRUE),
          Feeder→collect(r| r→adjt!=NULL AND r→adjt→type==SWITCH AND
                            r→adjt→operable=TRUE),
          Feeder→collect(r| r→AD→iterate(s| UnlockSwitches(s→pCmp, visited)),
                           r→OD→iterate(s| UnlockSwitches(s→pCmp, visited))),
          Feeder→iterate(r| Segment(r)→iterate(s| If visited→includes(s)==FALSE
                                                   Then UnlockSwitches(s, visited)
                                                   EndIf)
      EndIf)
```

Cotree Switch Functions

0. **CotreeSwitch()**
1. **FindOpSwt()**

Cotree Switch Globals

0. **seq of Component Loads**
1. **seq of Component Violations**

CotreeSwitch(Model M)

```
0.  set of Component Cotrees, seq of Component FP1, seq of Component FP2,
    set of Component openSwts
1.  M→collect(p| p→type==SWITCH)→collect(p| status=ON)
2.  Loads=M→collect(p| p→type==LOAD)
3.  Violations=M→collect(p| p→status==FAILED OR CheckConstraints(p)==FALSE)
4.  Loads→order(p<q if ( p→priority > q→priority OR
                  (M→exists(S→exists(t→AD→exists(u| u→pCmp==q) OR
                                   t→OD→exists(u| u→pCmp==q)))
                   AND
                   M→forall(S|S→forall(t|t→AD→collect(u| u→pCmp==p)→size==0 AND
                                      t→OD→collect(u| u→pCmp==p)→size==0)))))
5.  ClearPriorities(M)
6.  Loads→iterate(p| PropagatePriority(p))
7.  M→iterate(z| If Violations→size==0
              Then M→#index=M→size
              Else Violations→order(p<q if p→priority<q→priority),
                   FindOpSwt(Violations→first)→status=OFF,
                   ClearPriorities(M),
                   Loads→iterate(p| PropagatePriority(p)),
                   Violations=M→collect(p| p→status==FAILED OR p→f>p→c))
8.  Cotrees=M→collect(p| p→type==SWITCH AND p→adjt!=NULL AND
                   p→status==ON AND p→adjt→status==ON AND
                   FPT(p)→last!=FPT(p→adjt)→last)
9.  Cotrees→iterate(p| openSwts=FPT(p)→collect(q| q→type==SWITCH),
               openSwts=openSwts→append(FPT(p→adjt)→collect(q| q→type==SWITCH)),
               openSwts→order(q<r if q→f<r→f),
               openSwts→iterate(q| q→status=OFF,
                             If M→collect(r|CheckConstraints(r)==FALSE)→size==0
                             Then openSwts→#index=openSwts→size
                             Else q→status=ON))
10. AddLoops(M)
```

FindOpSwt(Component pCmp)

```
0.  set of Component candidates, int lowPrior
1.  candidates=FTtoB(pCmp)→collect(p| p→type==SWITCH)
2.  candidates=candidates→including(FPT(pCmp)→collect(p| p→type==SWITCH))
3.  If pCmp→type==SWITCH
    Then candidates=candidates→including(pCmp)
    EndIf
4.  lowPrior=candidates→collect(p| candidates→exists(q| q→priority<p→priority)==FALSE)→first→priority
5.  candidates=candidates→excluding(p| p→priority > lowPrior)
6.  candidates=candidates→excluding(p| candidates→exists(q| FPT(q)→includes(p)==TRUE))==TRUE)
7.  return candidates→collect(p| candidates→exists(q| q→f>p→f)==FALSE)→first
```

Adding Loops

AddLoops(Model M)

```
0.  seq of Component Underfed, set of Component CmpSeg={}, seq of Component SegSwts
    seq of Component Trace
1.  Underfed=M→collect(p| p→type==LOAD AND IsUnderfed(M, p)==TRUE)
2.  Underfed→order(p<q if ( p→priority > q→priority OR
                          (M→exists(S→exists(t→AD→exists(u| u→pCmp==q) OR
                                               t→OD→exists(u| u→pCmp==q)))
                           AND
                           M→forall(S|S→forall(t|t→AD→collect(u| u→pCmp==p)→size==0 AND
                                                  t→OD→collect(u| u→pCmp==p)→size==0)))))
3.  Underfed→iterate(p|
        If IsUnderfed(p)==TRUE AND
          FPT(p) →forall(q|Segment(q) →collect(r|r→status==FAILED)→size==0)==TRUE
        Then Trace=FPT(p),
             Trace→iterate(q|
             If CmpSeg→excludes(q)
             Then CmpSeg=Segment(q),
                  SegSwts=CmpSeg→collect(r| r→type==SWITCH AND r→status==OFF),
                  SegSwts=SegSwts→including(CmpSeg→collect(r→adjt| r→adjt!=NULL
                                                         AND r→adjt→type==SWITCH
                                                         AND r→adjt→status==OFF
                                                         AND Segment(r→adjt→fpt) →
                                                           collect(s|s→status==FAILED) →size==0)),
                  SegSwts→collect(r| r→status=ON),
                  If CheckConstraints()==TRUE AND IsUnderfed(M, p)==FALSE
                  Then SegSwts→order(r<s if r→f<s→f)
                       SegSwts→iterate(r| r→status==OFF,
                                     If IsUnderfed(M, p)==TRUE
                                     Then r→status=ON,
                                          SegSwts→#index=SegSwts→size
                                     EndIf)
                  Else If CheckConstraints()==FALSE
                       Then SegSwts→collect(r| r→status=OFF),
                            Trace→#index=Trace→size
                       EndIf
                  EndIf
             EndIf)
        EndIf)
```

IsUnderfed(Model M, Component pCmp)

```
0.  return (pCmp→f < pCmp→f_req) OR
        M→exists(p| p→AD→exists(q| q→pCmp==pCmp AND
                                   pCmp→f / pCmp→f_req < q→percent)) AND
        M→exists(p| p→OD→exists(q| q→pCmp==pCmp AND
                                   pCmp→f / pCmp→f_req < q→percent)
                  AND
                  p→OD→exists(q| q→pCmp→f / q→pCmp→f_req >q→percent)==FALSE)
```