OPTIMAL OPERATIONAL STRATEGIES FOR A DAY-AHEAD ELECTRICITY MARKET IN THE PRESENCE OF MARKET POWER USING MULTI-OBJECTIVE EVOLUTIONARY ALGORITHMS

by

DEEPAL RODRIGO

B.S. ENG., University of Moratuwa, 1986
M.B.A., University of Colombo, 1992
M.S., Kansas State University, 1995

AN ABSTRACT OF A DISSERTATION

submitted in partial fulfillment of the requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Electrical and Computer Engineering
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2007

Abstract

This dissertation introduces a novel approach for optimally operating a day-ahead electricity market not only by economically dispatching the generation resources but also by minimizing the influences of market manipulation attempts by the individual generator-owning companies while ensuring that the power system constraints are not violated. Since economic operation of the market conflicts with the individual profit maximization tactics such as market manipulation by generator-owning companies, a methodology that is capable of simultaneously optimizing these two competing objectives has to be selected. Although numerous previous studies have been undertaken on the economic operation of day-ahead markets and other independent studies have been conducted on the mitigation of market power, the operation of a day-ahead electricity market considering these two conflicting objectives simultaneously has not been undertaken previously. These facts provided the incentive and the novelty for this study.

A literature survey revealed that many of the traditional solution algorithms convert multi-objective functions into either a single-objective function using weighting schemas or undertake optimization of one function at a time. Hence, these approaches do not truly optimize the multi-objectives concurrently. Due to these inherent deficiencies of the traditional algorithms, the use of alternative non-traditional solution algorithms for such problems has become popular and widely used. Of these, multi-objective evolutionary algorithms (MOEA) have received wide acceptance due to their solution quality and robustness. In the present research, three distinct algorithms were considered: a non-dominated sorting genetic algorithm II *(NSGA II*), a multi-objective tabu search algorithm (*MOTS*) and a hybrid of multi-objective tabu search and genetic algorithm (*MOTS/GA*). The accuracy and quality of the results from these algorithms for applications similar to the problem investigated here reinforced the selection of these algorithms. The results obtained from each of the three algorithms used in the evaluations are very comparable. Thus one could safely conclude that the results obtained are valid. Three distinct test power systems operating under different conditions were studied for evaluating the suitability of each of these algorithms. The test cases included scenarios in which the power system was unconstrained as well as constrained. Repeated simulations carried out for the same

test case with varying starting points provided evidence that the algorithms and the solutions were robust.

Influences of different market concentrations on the optimal economic dispatch are evidenced by the *pareto-optimal-fronts* obtained for each test case studied. Results obtained from a traditional linear programming (*LP*) based solution algorithm that is used at present by many market operators are also presented for comparison. Very high market-concentration-indices were found for each solution from the *LP* algorithm. This suggests the need to use a formal method for mitigating market concentration. Operating the market at industry-recommended threshold levels of market concentration for selecting an optimal operational point is presented for all test cases studied. Given that a solution-set instead of a single operating point is found from the multi-objective optimization methods, additional flexibility to select any operational point based on the preference of those operating the market clearly is an added benefit of using multi-objective optimization methods. However, in order to help the market operator, a more logical fuzzy decision criterion was tested for selecting a suitable operating point. The results show that the optimal operating point chosen using the fuzzy decision criterion provides a higher economic benefit to the market, although at a slightly increased market concentration.

Since the main objective of this research was to simultaneously optimize the economic operation of a day-ahead market while ensuring minimal market power by individual generator owners, the proposed method is much improved from the current industry practice. The current practice of after-the-fact mitigation of market power has created various problems for both the market operator and the market participants, giving rise to a large numbers of disputes and resettlement activities. Hence, an approach that mitigates market power at the time of market dispatch as used in this research would bring about a more efficient market operation.

OPTIMAL OPERATIONAL STRATEGIES FOR A DAY-AHEAD ELECTRICITY MARKET IN THE PRESENCE OF MARKET POWER USING MULTI-OBJECTIVE EVOLUTIONARY ALGORITHMS

by

DEEPAL RODRIGO

B.S. ENG., University of Moratuwa, 1986
M.B.A., University of Colombo, 1992
M.S., Kansas State University, 1995

A DISSERTATION

submitted in partial fulfillment of the requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Electrical and Computer Engineering
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2007

Approved by:

Major Professor
Dr. Anil Pahwa

Abstract

This dissertation introduces a novel approach for optimally operating a day-ahead electricity market not only by economically dispatching the generation resources but also by minimizing the influences of market manipulation attempts by the individual generator-owning companies while ensuring that the power system constraints are not violated. Since economic operation of the market conflicts with the individual profit maximization tactics such as market manipulation by generator-owning companies, a methodology that is capable of simultaneously optimizing these two competing objectives has to be selected. Although numerous previous studies have been undertaken on the economic operation of day-ahead markets and other independent studies have been conducted on the mitigation of market power, the operation of a day-ahead electricity market considering these two conflicting objectives simultaneously has not been undertaken previously. These facts provided the incentive and the novelty for this study.

A literature survey revealed that many of the traditional solution algorithms convert multi-objective functions into either a single-objective function using weighting schemas or undertake optimization of one function at a time. Hence, these approaches do not truly optimize the multi-objectives concurrently. Due to these inherent deficiencies of the traditional algorithms, the use of alternative non-traditional solution algorithms for such problems has become popular and widely used. Of these, multi-objective evolutionary algorithms (MOEA) have received wide acceptance due to their solution quality and robustness. In the present research, three distinct algorithms were considered: a non-dominated sorting genetic algorithm II *(NSGA II*), a multi-objective tabu search algorithm (*MOTS*) and a hybrid of multi-objective tabu search and genetic algorithm (*MOTS/GA*). The accuracy and quality of the results from these algorithms for applications similar to the problem investigated here reinforced the selection of these algorithms. The results obtained from each of the three algorithms used in the evaluations are very comparable. Thus one could safely conclude that the results obtained are valid. Three distinct test power systems operating under different conditions were studied for evaluating the suitability of each of these algorithms. The test cases included scenarios in which the power system was unconstrained as well as constrained. Repeated simulations carried out for the same

test case with varying starting points provided evidence that the algorithms and the solutions were robust.

Influences of different market concentrations on the optimal economic dispatch are evidenced by the *pareto-optimal-fronts* obtained for each test case studied. Results obtained from a traditional linear programming (*LP*) based solution algorithm that is used at present by many market operators are also presented for comparison. Very high market-concentration-indices were found for each solution from the *LP* algorithm. This suggests the need to use a formal method for mitigating market concentration. Operating the market at industry-recommended threshold levels of market concentration for selecting an optimal operational point is presented for all test cases studied. Given that a solution-set instead of a single operating point is found from the multi-objective optimization methods, additional flexibility to select any operational point based on the preference of those operating the market clearly is an added benefit of using multi-objective optimization methods. However, in order to help the market operator, a more logical fuzzy decision criterion was tested for selecting a suitable operating point. The results show that the optimal operating point chosen using the fuzzy decision criterion provides a higher economic benefit to the market, although at a slightly increased market concentration.

Since the main objective of this research was to simultaneously optimize the economic operation of a day-ahead market while ensuring minimal market power by individual generator owners, the proposed method is much improved from the current industry practice. The current practice of after-the-fact mitigation of market power has created various problems for both the market operator and the market participants, giving rise to a large numbers of disputes and resettlement activities. Hence, an approach that mitigates market power at the time of market dispatch as used in this research would bring about a more efficient market operation.

Table of Contents

List of Figures

List of Tables

Glossary of Terms

1. ANN – Artificial Neural Network
2. ADHHI – Average Dynamic Herfindhal-Hirschman Index
3. Bus – An electrical connection point where physical electrical devices are connected
4. Crossover – a term commonly used in Genatic Algorithms to represent the formation of a child chromosome from a pair of parents
5. Day-Ahead Market – A forward electricity market for the 24 for the next operating day
6. DHHI – Dynamic Herfindhal-Hirschman Index
7. FERC – Federal Energy Regulations Commission
8. Generator – A physical device that is used for generating electricity
9. Generation – The output from a generator
10. HHI - Herfindhal-Hirschman Index
11. LR – LaGrangian Relaxation Algorithm
12. LP – Linear Programming
13. Load – A physical device that consumes electricity
14. LMP - Locational Marginal Price
15. MW – Megawatts
16. $/MWh – Cost measured in dollors per mega watt hour
17. MOEA – Multi-Objective Evolutionary Algorithms
18. MOTS – Multi-Objective Tabu Search Algorithm
19. MOTS/GA – Multi-Objective Tabu and GA hybrid Algorithm
20. Market day – The 24 hour period an electricity market is operated
21. NSGA II – Non-Dominant Sorting Algorithm
22. RTO – Regional Transmission Organization
23. SA – Simulated Annealing
24. Time horizon – the period that is considered in a market evaluation
25. TS – Tabu Search Algorithm

Acknowledgements

First and foremost let me begin by thanking my major professor Dr. Anil Pahwa for his kindness, unwavering support, patience, constant encouragement and guidance to make me complete this research. At this point I must also acknowledge the help I received from my committee members, Dr. Medhat M. Morcos, Dr. Prakash Krishnaswami, Dr. John E. Boyer and Dr. Robert B. Burckel for their support during my program of study. Special thanks goes to Dr. Sanjoy Das who took time to review and provide guidance to enhance the work undertaken in this research with his expertise in the area of evolutionary algorithms as well as Dr. Burckel for taking the time off his busy schedule to review and provide feedback to enhance the presentation of my dissertation.

I would also like to mention the constant encouragement I received from my mother Lalani, my sister Champa, my mother-in law Yasawathie, my sister-in-law Chitra, my brother-in-law Bandula and various other family members. I deeply appreciate your support throughout this journey.

Last but not least, let me thank my wife Udeshika and my daughter Ovini for accommodating and tolerating my busy schedule working on this research while working full time and never complaining when I am not there for family time. They have given the best support, encouragement and patience a husband and a father could hope for.

Dedication

I would like to dedicate this work to all my family members especially to my beloved deceased father and father-in-law, who I am sure, would have been very proud of my accomplishment.

CHAPTER 1 - Introduction

With the Federal Energy Regulation Commission's (*FERC*) initiative to deregulate interstate electricity trading, the need for all interacting parties to work more closely to meet the conditions stipulated by the *FERC* becomes paramount. Power marketers, generator owners, transmission owners, companies bidding for electrical power, and regional transmission organizations responsible for operating electricity markets interact with one another when participating in electricity trading activities. In this context a market operator is responsible for offering multiple energy commodities to the market participants so that they can effectively participate in the electricity market. It is customary for most markets to offer opportunities to the participants to submit their generator offers and demand bids ahead of the market day. To cater to the needs of the market, two market products are typically offered: the offering of a day-ahead market product and the offering of an hourly-market product. As the name implies, the day-ahead market allows market participants to offer and bid for a given 24-hour period, ahead of the actual market day. The hourly market then is expected to mitigate short-term energy short-falls or changes subsequent to their day-ahead offerings and bids. Since a majority of the market transactions are expected to take place in the day-ahead market, the study of strategies for solving a typical day-ahead market provided the motivation for this research. Traditionally, security-constrained-economic-dispatch and unit-commitment algorithms are used to match all generator offers with demand bids. The objective of these algorithms is to minimize the overall market operational cost subject to various power system constraints.

However, as mandated by the *FERC*, while the individual companies strive to reap maximum individual profits, the market operator has a responsibility to ensure that all parties in the market have equal opportunities to participate. In their pursuit of individual profit maximization, many market participants use the geographical and strategic location of their assets and market share to "game the market" or use their strategic position in the market to their advantage. This behavior has forced the market operator to take appropriate mitigating measures to contain adverse market-participant behavior. At present, mitigation of market-power behavior by market participants is accomplished by utilizing after-the-fact methods. Also many of the

current methodologies do not incorporate the impacts of the power system conditions into their evaluation. The majority of these methods use individual historical reference-prices of generators adjusted by fuel-cost variations ignoring the impacts of the condition of the power system. Some of the large energy markets in the United States that use these approaches of mitigation, show significant operational challenges to the market operator as well as the participants. Based on these observations, it is evident that an improvement in current mitigation strategies is needed. A simultaneous solution methodology that operates the day-ahead market while minimizing market power shows great promise at this juncture. Evaluation of simultaneous solution methodologies capable of accomplishing the desired enhanced operation of a day-ahead market and guidance in making operational decisions based on the results is presented in this dissertation.

The following section presents a wide catalogue of previous work undertaken in this area. It provides an important basis for demonstrating the significance and relevance of the research discussed and presented in detail in the ensuing chapters.

1.1 Literature Review

The whole area of market operations starting from the mid-eighties has evolved into one of the most popular areas on which researchers have focused attention. The fundamental concepts of location-based pricing of transmission services was first proposed in [1] and was presented at many international forums such as [2], [3]. It is evident that many researchers attempted to align known economic models with electricity market operations. Many considered oligopolistic models to represent the electricity market [4], [5]. Others proposed economic Nash equilibrium models for electricity markets [6], [7]. Features and definitions of oligopolistic markets, markets in Nash equilibrium, Cournot [8], and Bertrand market models show the importance of identifying a model which closely represents the day-ahead market. Some of the previous studies including work done by Song, *et al*. [7] considered simplifying assumptions such as each bidding company having perfect and complete information about every bidder in the market, and generators being responsible for paying for systems losses. In [6], a real-time market is evaluated assuming that each market participant can estimate his competitor's behavior. This appears to be a more realistic model and therefore is considered in the studies undertaken in this dissertation. Since one of the most critical requirements of a market operator is

mitigating market power for fostering unbiased operations, the operational strategies proposed here will focus on minimizing the market power of each company, while maintaining typical operational constraints. This is a new approach to the day-ahead market operational methodology. Even though some researchers have proposed new methods for market operations based on economic models and some others have proposed market-power monitoring methods [9], [10], [11], the literature review conducted did not reveal any previous work that combined limiting the market power while simultaneously arriving at an optimum operating strategy for a day-ahead electricity market. Hence, this research is expected to address a critical need of the market and formulate a unique approach for obtaining operational strategies.

Finding the optimal operational strategy for the day-ahead market is a complex optimization problem, in which the bids of generators and the loads are matched at the minimum overall market expense. A survey of the literature on work undertaken previously has highlighted a wide variety of methodologies for solving this very complex problem. Extensive work carried out by Ranatunge, *et al*.[12], and Dekrajangpetch, *et al*.[13] using linear programming techniques, Sugianto, *et al*. [14] using fuzzy classification models, de la Torre, *et al*.[5], Galliana, *et al*. [15], Sonmez, *et al.*[16], using nonlinear mixed-integer optimization algorithms, and Ptetridis, *et al*.[17] using evolutionary algorithms, show the importance of the problem. It is also evident that most of the traditional solution algorithms, as found in the work done by Ranatunge, *et al.*[12], and Dekrajangpetch, *et al*.[13] either convert the multiple-objective functions to be optimized into a single objective function via weighting schemas, or they optimize one objective function at a time. The fact that these approaches are not true concurrent-optimization methods led to the consideration of alternative non-traditional algorithms, which have become very popular due to their concurrent optimization capabilities. Moreover, evolutionary algorithms are found to perform well on optimization problems with complex optimality solution surfaces, which provides further justification for using such algorithms in this research. Work done by King, *et al*. [18], uses a multi-objective evolutionary algorithm for achieving economic dispatch taking the environmental impacts into consideration. Although this approach is somewhat closer to the approach considered in this dissertation, the latter considers a different solution space altogether. As in [18], a non-dominated sorting multi-objective genetic algorithm, known as *NSGA II*, developed by Deb, *et al.*[19] was selected as one possible suitable solution algorithm.

Successful application of a tabu search algorithm and hybrid tabu search/ genetic algorithm in solving similar problems, as done by Galloway, *et al.* [20], and Ongsakal, *et al.*[21], as well as problems in other industries as depicted in the work by Zdansky, *et al.*[22] also provide additional justification for selecting these algorithms.

After expanding the literature survey further, it became evident that many of the previous works concentrated on finding an optimal solution at a given time point; this is evident in works by Dekrajangpetch, *et al*. [13], and Keshav, *et al*.[20]. However, the reality is that the day- ahead market should cover all 24-hours of a given market day. This further complicates the problem, since not only will the solutions have to be optimal for a given hour, but also be able to provide an overall optimality in terms of how the generators are selected to feed into the market and how often they are turned on/off in given 24-hour period.

The research by King, *et al.*[18] has solved a simple market dispatch problem for a very small test system consisting of 3 generators. In [23], an optimal day-ahead network-constrained market problem is solved using the interior-point method for a 14-bus test system. In [24], although there is reference to evaluation of Locational Marginal Prices in a competitive market, the influence of market power as undertaken by this research is not considered. Also it is notable that the simulations in [24] are limited to being applied to a 14-generator test system. Looking further, it is evident that other researchers have investigated the possibility of combining multiple algorithms to improve performances of each algorithm operating in isolation. The work done by [20] and [21] uses a hybrid genetic algorithm and a combined simulated annealing /genetic algorithm to solve a similar economic dispatch problem successfully. To observe the suitability of such non-traditional solution algorithms for solving complex multi-objective problems, the literature survey encompassed studies carried out using combination algorithms and other multi-objective solution techniques. Merits of the multi-objective tabu search algorithm, as demonstrated by the work done by Ramirez-Rosado, *et al*. [25], prompted the selection of this algorithm as a potentially viable candidate for the work undertaken here. In [26], an investigation into integrating genetic algorithms with tabu search and simulated annealing for a unit commitment problem is presented. In [20] and [21] the use of hybrid genetic algorithms for solving similar complex problems is presented. This stimulated the interest in combinatorial non-

traditional algorithms and eventually was chosen as the third solution algorithm for investigation. From the studies done previously, the use of domain knowledge as a key contributor to the success of tabu search algorithm is noteworthy. Considering these merits, the multi-objective tabu search algorithm was thought to be a strong contender, worthy of being investigated.

Further analysis of the problem being solved presents the challenge of selecting an optimal operating point from the *pareto* non-dominant solution-set found from multi-objective optimization. From an extensive literature review it became quite evident that this was a challenge faced by all researchers solving multi-objective optimization problems. In recent years, many researchers have focused their attention on finding approaches for multi-criteria decision making. The work by [27], [28], [29], [30], and [31] provides many-criteria-decision-making approaches that have become popular. From these methods, the fuzzy-decision-criteria method proposed by Farina, *et al*. [29] was selected and tested for its suitability in this dissertation.

1.2 Summary

A summary of the organization of the succeeding chapters of this dissertation is described in this section. Chapter 2 discusses the formulation of the mathematical model that represents the day-ahead market operations. The nature of the transmission systems, and physical system constraints that impact market operations were incorporated into this model. Chapter 3 focuses on the investigation of economic models and their applicability to represent a day-ahead market. Impacts of market power, methods for monitoring and mitigation are presented in Chapter 4. Chapter 5 presents a survey of algorithms that are deemed suitable for solving the day-ahead market dispatch and market-power-mitigation problems, while in Chapter 6 the approaches to modeling the day-ahead market operation problem using the three proposed multi-objective evolutionary algorithms are discussed. This Chapter also presents a mathematical basis of fuzzy-decision-making for selecting a better operational point from the non-dominant solution set, rather than using industry recommended threshold values. Chapter 7 presents results from all case studies applied to three distinct test systems. Results from different algorithms under different operational conditions are also compared with one another. Finally, Chapter 8 presents conclusions derived from the research. This chapter also outlines possible avenues for future research.

CHAPTER 2 - Day-ahead Market Problem

In the present context, a day-ahead market will mean a 24-hour period starting from the hour 0 running through the hour 23 of a market day. Typically the market day is the next calendar day from the day the evaluations are done. In all markets that offer this commodity, the dispatch schedules for the day-ahead market are compiled hours ahead of the beginning of the operating day-ahead day. Part of the day-ahead market participation rules are that all generators and loads willing to participate in the day-ahead market have to supply their offers and bids to the market operator by a pre-specified deadline. In most markets this time is set at 9 A.M., which is15 hours ahead of the beginning of the market period. This lead-time is allowed for solving the security-constrained economic dispatch and unit-commitment for the 24-hour period, and for the market participants to accept or rebid for the day-ahead market on the basis of first published results. Once the market operator receives these offers and bids by the deadline, they are considered as viable, and will be considered in the optimal security-constrained economic dispatch and unit- commitment. Even though the load bids and generator offers will be done as integrated hourly values for every hour of the day, the market is considered continuously operating for the entire 24-hour period transitioning smoothly from the previous day to the current market day. Hence, a routine that considers the whole 24-hour period rather than each individual hour in isolation will become more realistic and representative. It is evident that this is a very special economic dispatch of generating units for every hour of the day, which may bring new generators online whenever needed. Based on the above, the day-ahead market dispatch objective function can be formulated mathematically as follows.

t	time index, this represents each hour block of the day-ahead day.
T	optimization horizon, under the present context this spans a 24-hour period
i	generator index
I	generator set that participates in the market
$S_i(t)$	MW (=megawatts) quantity offered by generator i into the market at hour t
$\alpha_i(t)$	price in \$/MW received by the generator i, for supplying $S_i(t)$ MW at hour t

j load index

J load set that participates in the market

$D_j(t)$ MW load demand by load j at hour t into the market

$\beta_j(t)$ price in \$/MW to be paid by the load j for $D_j(t)$ MW at hour t

Now considering the hour t, the objective of the market should be to match the total load demanded from the market with the generation supply offers available to the market while determining an optimal dispatch schedule where the total cost to operate the market is kept at a minimum. In other words, load demand of the operation region under consideration will be met by using the most economical and available generators without violating physical line or other reliability and operational limitations of the operational region. Based on this rationale, the objective function solved for hour t would be,

$$\left\{ \sum_{i \in I} \alpha_i(t)\, S_i(t) - \sum_{j \in J} \beta_j(t)\, D_j(t) \right\}$$

Considering the 24-hour day-ahead operation period, the objective function to be found becomes,

$$\left[\sum_{t=0}^{T} \left\{ \sum_{i \in I} \alpha_i(t)\, S_i(t) - \sum_{j \in J} \beta_j(t)\, D_j(t) \right\} \right]$$

However, the problem at hand is constrained due to the fact that the number of generators available to supply the market is limited and generators as well as other power system components have physical operational limits. Considering the nature of electricity markets and physically interconnected power systems, the objective function will be subject to the following operational constraints.

2.1 Bus Power Balance Constraints

In order to facilitate solving the above optimization problem, linearized power-flow model also known as “DC loadflow” will be used. Considering the nature of a typical power system, one can write a power balance equation to represent the fact that net power injected at a

given bus (= a connection node) should be the same as what is being withdrawn. This can be symbolically represented as follows:

$$P_i(t) = \sum_{\substack{l=1 \\ and \\ fr(l)=i}}^{NL} BFlow_{l,t} - \sum_{\substack{l=1 \\ and \\ to(l)=i}}^{NL} BFlow_{l,t}, \; i = 1,2,...., NL,$$

where $P_i(t)$: Net MW injections into bus i at hour t;

NL: Number of buses in the system;

$fr(l)$: from-end of branch l;

$to(l)$: to-end of branch l;

$BFlow_{l,t}$: Power flow over branch l at hour t.

2.2 Branch-Capacity Limit Constraint

Based on the thermal and other physical limits of transmission lines, each transmission line in a power system will have a capacity limit. When operating the power system, these capacity limits are expected to never be exceeded. This requirement can be represented mathematically as:

$$\left| BFlow_{l,t} \right| \le BFlow_{l,t}^{\max},$$

where $BFlow_{l,t}^{\max}$: is the capacity limit in MW of branch l at hour t.

2.3 Generator Ramp-Rate Constraint

The MW output change in a generating unit from one hour to the next is limited by the maximum-ramp-rate of the generating unit. Hence, if the output from a generator i at time t is given by $S_i(t)$, then in order to ensure that the change in generation from one operational state to the next does not exceed its ability to ramp up or down, the following constraint can be defined:

$$S_i(t\text{-}1) - S_i(t) \le RampRateMax_i(t); \; t=0,1,2\ldots23,$$

where $RampRateMax_i$ (t) is the maximum ramp rate of unit i at hour t.

2.4 Economic Minimum and Maximum Operating Constraint

All generators participating in the market are required to be operated so that their economic maximum and minimum operating limits are maintained. Also, since the power system is expected to retain its ability to recover from the largest contingency in the system, adequate reserves need to be maintained. Thus, the contributions from each generator to the reserve requirements of the market need to be included. The relationship of each generator to its economic maximum limit can be represented as:

$$S_i(t) + S^{spin}{}_i(t) \leq EcoMax_i(t).$$

Similarly all generators are expected to be operated above their minimally economic limits:

$$S_i(t) \geq EcoMin_i(t),$$

where $EcoMax_i(t)$: Economic maximum limit of unit i at time t,

$S^{spin}{}_i(t)$: Spinning reserve dispatch of unit i required at time t,

$EcoMin_i(t)$: Economic minimum of unit i at time t.

2.5 Operating-Reserves Requirement Constraint

To ensure that the power system can recover from an unplanned contingency, a pre-specified amount of operating reserves for the system needs to be maintained. This system operating requirement is then converted into corresponding individual contributions from each of the generators supplying energy to a given power system. The relationship of the MW output of a generating unit to its operating-reserve obligation can be represented in the following manner:

$$S_i(t) + S^{oper}{}_i(t) \leq EmergMax_i(t)$$

where $EmergMax_i(t)$: Emergency maximum limit of the unit i at time t,

$S^{oper}{}_i(t)$: Operating reserve dispatch of unit i required at time t.

2.6 Generator on-line/off-line Switching Constraint

In the solution, movement of generators in and out of the dispatch must be minimized to avoid the need to keep generators running ready to supply the market even when it is uneconomical to select these generators. In this approach, when the system load demand increases, all generators that were generating during the previous time period excluding those that have reached their maximum capacity will be chosen first. If on the other hand the system load decreased, only a subset of generators which were already supplying the market will be chosen to reduce output on their economic merit. This pre-selection process is extended to cover the entire study horizon to ensure that a minimum number of generators are selected for operating the market. Since the dispatch from the previous hours is always considered for every subsequent evaluation, the day-ahead market transitioning from the previous day to a new day will always be smoother than an approach that does not consider the effects of the previous hours.

The above formulation considers that the offers and bids made by participating entities would faithfully follow the basic market principles. However, in real life it is a well known fact that all profit-driven entities will always attempt to maximize their benefits at every opportunity. By the same token, all other parties who are victimized by such influencing would strive for a market that treats every participant equally. Given this, the market operator is challenged and is responsible for ensuring that such undue influences are minimized for the greater good of the market. In order to represent a realistic and unbiased market-operating process, market power influences will have to be eliminated while conducting economic-dispatch and unit-commitment for the market day. Before implementing a market-power mitigation strategy into the economic solution, one has to first identify the basic nature of the electric industry operating a day-ahead market. The next section of this research will therefore concentrate on aligning an economic model to the day-ahead market operations to better understand the influences imposed by various constituents of a day-ahead market. The chapter following proposes a practical market-mitigation scheme to be used effectively based on the alignment of an economic model with the day-ahead electricity market.

CHAPTER 3 - Economic Models for a Day-Ahead Electricity Market

The main objective of this section is to find out which economic model best represents the day-ahead electricity market. Looking at the nature of the business, one can say that there are multiple generator-owning companies that compete with each other for a share of the same market. For the most part, all these companies share the same geographic and economic boundaries. For example, only those generators that are within the market region will be able to offer generation bids. They all compete for one homogeneous commodity, electric power, and all have the same objective which is to maximize their own net benefits from the market. Moreover, all competing generator-owning companies anticipate that rival firms will also be competing for the same market share. Based on the consumer demand for power, the generator-owning companies will decide how much to generate for a given period so that their profits can be maximized. It is a well-known fact that electric power has no shelf life, making this commodity extremely perishable. Also, the ability to influence the market by a given company will depend on the number of generators and the relative locations of the generators it owns. In the present context, the market is managed by a regional transmission organization that is expected to take adequate measures to mitigate market power and offer a fair market to the participants.

Considering all these factors prevalent in a day-ahead market, it would be a worthwhile exercise to look at some economic models that have evolved in game theory. The next step would be to evaluate each of these models with respect to the core features of a day-ahead electricity market.

As presented and analyzed by game theory, a market that exhibits oligopolistic behavior shows a complex series of strategic moves and reactive countermoves among rival firms. In an oligopolistic market, firms are assumed to anticipate rival actions. Also, all competing firms in the same market try to supply a homogeneous consumption good. The consumer side is represented by a fixed demand function in such a market. The firms decide how much to produce

of a perishable consumption good, and they decide upon a number of information signals to be sent into the population in order to attract customers. Considering the nature of the day-ahead energy market, one can see that all generating companies are competing to supply a specified MW amount of electricity as demanded by the end-user. The features of a homogeneous commodity, fixed demand for a given time period, and anticipation of rival actions by competing companies are all present in the day-ahead market and are in line with the core characteristics of an oligopolistic market [5]. However, in a Regional Transmission Organization (henceforth RTO) controlled market place, the generating companies are expected to generate what they are asked to, rather than to decide on their own how much to generate; and these companies are expected not to exercise market manipulation strategies. Investigating further, some oligopoly models allow for a few large firms to behave as oligopolists and a fringe of small firms to behave as competitors. Each oligopolist believes that its actions influence price, and it must consider the reaction of the other oligopolists when making decisions. It is known that price competition leads to price wars, which may cause some producers to exit the market or merge with other producers to strengthen their position. Looking at the history of the electricity markets, one observes these trends. In some typical oligopolistic markets, firms use product differentiation and advertising campaigns to gain competitive advantages over their rivals. This however, is not a feature in the electricity markets. The product offered by every company is the same and firms have no way of differentiating the product offered for sale. Thus, the electricity market can be considered as a specialized oligopolistic market. Another feature found in an oligopolistic market is its expectation that any new entrants to the market will have a large market share after entry. This condition must be satisfied before they are allowed to enter the market. The next feature of such a market assumes that only economic barriers would prevent new participants from entering the market. The first condition does not seem to be applicable to electricity markets. This is because any independent power producer who wishes to participate in a market can enter the market without having to hold a major market share. The second condition, however, can be considered appropriate for the day-ahead electricity market operated by a centrally managed regional transmission organization.

Another major feature of an oligopoly is that each firm considers its opponents' response to any strategy it contemplates. This is true in any market scenario, in an electricity market or

elsewhere. All firms entering a profitable market have to contemplate the opponent's action if they are to remain successful in the market. As is evident, there are many variants of an oligopoly market. Many companies choose price as their primary decision variable, while many others choose quantity offered for sale as their primary decision variable. As far as the electricity markets are considered, price can be considered as the strategic decision variable. However, there could be instances where the quantity also becomes the driver. For example, when the power system is congested, a company that has generation capacity locally to meet the needs of a given load will enjoy its strategically advantageous location in the market over another company which has generation capacity at a different geographic location. Variations to the oligopoly economic model are available in the literature for different operational scenarios. In some cases, all firms are assumed to be making simultaneous decisions. In other situations a sequential-move pattern is assumed. Both these scenarios can be considered to be applicable to electricity markets. Depending on the positioning of certain firms in the electricity market, some may become involved in simultaneous decision-making. By the same token, there could be others which would follow a leader, or be a leader in the market.

Market environments may also differ with respect to the assumed time horizon. Some models assume a single decision period while others assume multiple or infinite time periods. In the day-ahead market it is expected that all firms participating consider the same time period, a single 24-hour day. Finally, market environments may also differ with respect to the amount of information each firm has: all firms could have the same information or some firms could have more complete information than others. This scenario is very representative of the electricity market. Vertically-integrated electric utilities sometimes have access to information that non-traditional, non-integrated firms don't have.

Participants in an oligopoly market can exercise either little or great market power. Intense price competition between two or more firms can erupt and dissolve market power in short order. However, firms may recognize that intense price competition is painful and therefore would prefer to engage in cooperative and non-cooperative collusion that facilitates a shared exercise of market power. Historical observations on the electricity markets have shown companies colluding from time to time for their mutual benefit. A variation of collusion is found

in what is known as the Stackelberg strategy [4], [32]. According to the Stakelberg model, two sets of market participants are considered; market leaders and market followers. Market leaders make decisions based on the reactions of the market followers. Market followers on the other hand without recognizing how their decisions affect the leaders', attempt to follow the actions taken by the leaders. Market responses as described in the Stackelberg strategy as well as cooperative and non-cooperative collusions are possible in electricity markets. The challenge for the market operators then would be to ensure that these strategies are properly mitigated.

Assuming that in an *RTO* controlled market there is no room for market power, the generators will therefore become pure price takers. They will honor the price that is set by the central dispatch entity or the *RTO*. This model equates to pure competition or the well-known Bertrand economic model [4]. Even though every *RTO* strives for this kind of a market, this model is of only academic interest and is hard to achieve in practice. Trying to generalize this behavior, economists have come up with an alternative model called a generalized Bertrand strategy. In this model, the quantity delivered by one company will depend on the price asked by the company itself as well as the prices asked by other competitors in the market. However, every company in the market place assumes that the others will not alter their prices based on what another company offers. It also assumes that if the price quoted by a given generating company is the lowest offered among rival producers, that generator will be selected to deliver up to its economically maximum limit before any other generator is chosen. If on the other hand the company does not ask the lowest price, it is further assumed that it will not be selected to supply any power to the market. While some of the considerations in this model are applicable to day-ahead markets, the fact that companies are assumed to be nonresponsive to pricing strategies of others is definitely not applicable.

The next economic market strategy is based on gaming in quantities; this model is commonly identified as the Cournot strategy [4], [6], [7]. In it, the price asked by a given company will be based on the quantity supplied by it as well as the quantities supplied by the others. However, the model assumes that the quantity supplied by each company will be a fixed amount. In an *RTO* controlled market this model will not be applicable for the most part; in this scenario, unless each participant resorts to manipulating the market by holding back the quantity

supplied, forcing a supply deficit in order to gain higher prices, the supply by each company is deemed to be fixed. From time to time companies do resort to these sorts of market manipulation strategies.

Depending on each market participant's choice, many of the previously discussed market conditions could exist in a day-ahead market. If companies in the market adopt a Bertrand strategy, the market will see pure competition. If on the other hand companies adopt a Cournot strategy, they might see individual benefits depending on the market conditions, with negative consequences to those not adopting the Cournot strategy. At times some companies also participate in cooperative and non-cooperative collusion strategies. Considering all possible behavior patterns that could be adopted by the market participants, an unbiased market clearing process must be chosen that restrains the market participants from exercising undue market manipulation. Chapter 4 of this dissertation presents methodologies for identifying, measuring, and preventing companies from resorting to gaming or exercising undue market power.

CHAPTER 4 - Market-Power Identification and Mitigation

With the evolution of a single market based on price signals, the price of electricity in the market will be determined by the cost of the most expensive plant scheduled to operate and fulfill the market demand. This unit is commonly called the marginal unit, and a given market could end up having a single marginal unit that results in a single Locational Marginal Price (*LMP*) for the market or multiple marginal units and multiple *LMP*'s if the market is congested. The basis for deciding on the marginal unit and thereby the *LMP* is a centralized dispatch conducted by a regional transmission organization. In such a centralized dispatch, all generators bidding into the market will receive the same price based on the marginal unit. This causes some companies to receive more than what they asked for, while the generating company that supplied the marginal amount of power or the unit that sets the price will receive the market-clearing price, or the cost of production of that generator. Based on this principle, generator owners can deduce that unless they offer their generators at an economically attractive price they could potentially be replaced by lower offers from their competitors. Consequently, the generator owners also know that they will receive the price at which the market was cleared even though the price they offered to the market was lower. Under this scenario the companies whose generators were offered at a below the market-clearing price would make a profit. Given these two scenarios, one can see that it is rather risky to try to manipulate the market when the load on the system is closer to the base load. This is simply because there is an abundance of generators that are available at these load levels and hence if a generator owner tries to inflate the price, he would probably be replaced by another, thereby losing his ability to participate in the market altogether. However, as the system demand increases, more and more units will be selected and a fewer number of generators will be left for supplying the load demanded. The remaining generators will be available at a higher cost to the market. If one company held back its generation at lower load conditions with the hope of offering the same generator at a higher price when the load was higher, much above its production cost, the company would clearly make a higher profit. This behavior however, is not encouraged since the overall price to the market will increase in this instance and the company adopting this strategy is manipulating the market to maximize its profits.

Looking closer, one can see that companies that own a large number of generators which are available to participate in a given electricity market have the ability to create such artificial conditions. History confirms instances where market regulators had to resort to enforcing mitigation strategies, such as placement of "market caps", to control these kinds of behavior [32]. While "market caps", is one mechanism for controlling the market, it fails to give any consideration to the operational conditions prevalent in the market. Clearly, the prices paid by loads in a given market will be driven by the congestion present in the market. Hence, a market power identification and mitigation strategy that incorporates the existing power system conditions is needed.

According to the guidelines set forth by the US Department of Justice, market concentration of a company is evaluated using an index identified as the Herfindhal-Hirschman Index [33]. This index is defined as follows:

$$HHI = \sum_{i=1}^{N} s_i^2$$

Here, the summation considers all N participants in the market and s_i refers to the market share of each participant. The market share of each participant is typically expressed as a percentage. Hence, the maximum *HHI* that can result in a given market is 10,000. This occurs when one company owns the whole market or has the market share of 100%. That company has complete control over the market, there is a complete monopoly. On the other hand, if a given market has 10 different companies each of which has 10% market share, the resulting *HHI* will be 1000. Since clearly none of the companies has any dominance over the others in terms of market share, none will be able to influence the market. The US Department of Justice divides the *HHI* into three ranges. When the index is below 1000 the market is considered unconcentrated. When the index is between 1000 and 1800 the market is considered moderately concentrated, while the market is considered heavily concentrated when the index is above 1800. According to the guidelines, the market share percentage is defined for a given market horizon. Therefore, determination of this index in a congested electricity market is not straightforward.

This is due to the fact that the generators and the quantity they would be supplying to the market from one hour to the next tend to change, depending on the power system conditions. The evaluation becomes even more complicated when the units owned by a given company are geographically dispersed. Under congested power system operation conditions, these generators would belong to separate operational islands making it difficult to identify the possible market manipulation exercises. Moreover, the evaluations need to pay attention to the amount of capacity left in each generating unit over and above its current dispatch level, along with those generators that have exhausted their capacity [34]. As discussed in [35], market power exercised by participants in a day-ahead electricity market is described as demonstrating horizontal market power. Here the behavior is seen to be prompted by the nature of the market-clearing methodology in which all generators chosen for dispatch are paid the market-clearing price. In many cases these generators will be paid a price above their production costs.

Looking closer at the day-ahead market scenario, one would expect the load profile throughout the day to vary and follow the typical system load profile, which has peaks of high and valleys of low load demand. As expected, at the base-load levels almost all the generators are able to participate in the market and any one company trying to inflate the price will risk being replaced by another generator from a competitor. However, as the system demand goes up, more and more generators from the pool of available resources will be selected and fewer and fewer generators will be left with available capacity to offer into the market. At this point, only those generators which are more expensive will be left, since the overall approach to market operation is to select the most economical generators first, followed by the next most economical set of units, and so on. Given this, the use of the standard *HHI* to measure market concentration seems inappropriate for the day-ahead market. It is clear that a modified index that represents the dynamic nature of the market has to be used. In this formulation only those units that have capacity remaining to offer at a given price-level along with information on ownership share of each generator-owning company will be considered in arriving at the market-share index. Formulation of this modified Dynamic Herfindhal-Hirschman Index (*DHHI*) can be represented as:

$$DHHI = \sum_{i=1}^{N(D)} [s_i(D^+)]^2 .$$

Here *N(D)* is the number of companies with capacity still left to offer to the market, and $S_i(D^+)$ represents the revised market-share of generator owner *i,* with capacity still left to offer into the market. Similar to the static *HHI*, since the % market share is considered in calculating the *DHHI*, the maximum possible value will be 10,000 with this formula as well. Due to its closeness to the static *HHI*, a logical approach when using the *DHHI* is to use the same three ranges recommended by the Department of Justice.

The merits of using the modified *DHHI* are best explained by an example. Let's assume that there are 10 generators offered into a given day-ahead market. Out of these, assume that a single company owns 3 generators, while 7 distinct companies own the remaining 7 generators. Also, to simplify the analysis, suppose the capacities of the generators are 100MW each. If a static *HHI* were calculated for this scenario, an index of 1600 = (30^2+7*10^2) is found. Since this value is below the high concentration threshold guidelines provided by the Department of Justice, one could inadvertently conclude that the market is only moderately concentrated. However, when the scenario is such that the system load was 600 MW at the last hour and the present hour has additional load demands to be met, the generators with capacity left to offer will be expected to provide the increased load. Assuming that at this operational condition only 1 generator from the company which owns the 3 generators has been fully dispatched and 5 other generators are also fully dispatched, the market share of the first company now becomes 50%, with the remaining market share of 50% divided equally among the independent companies. Using these market-share numbers, the *DHHI* for this hour becomes 3750 = (50^2+25^2+25^2). This clearly demonstrates the true market power of the first company and that it could potentially game-the-market by using its strategic position. At an index value of 3750 the market now is in the highly concentrated region. Therefore, the market operator has the appropriate signal to mitigate this condition by adopting a more equitable dispatch arrangement that does not allow companies this sort of market advantage.

Predictability of market conditions plays an important role in a company's ability to manipulate the market to its benefit. If the load profile for the market horizon can be predicted,

individual companies can predict the bidding patterns of other generators in the market. As the load increases, the number of generators available to the market becomes more limited and this prediction becomes easier. If the market auction is repeated frequently, the predictability of the market conditions becomes even easier. Based on these observations, it is seen that the real-time markets that are closer to the operational window are easier to predict and provide additional opportunities for manipulation. Because the day-ahead market is cleared once on the previous day, opportunities for participants to predict possible outcomes of such a market decrease. Hence it is expected that efforts to manipulate the market by its participants will be less likely to take place, although they are still possible in day-ahead markets. However, since market participants always strive for opportunities for their individual benefit, any mitigating strategies undertaken at the time of day-ahead market dispatch would become very effective.

Generally there are two mechanisms that a company adopts in manipulating a given market: withholding some fraction of capacity that can be offered, and offering some fraction of its capacity at a price markedly above the marginal price. Let us assume that all generators bid their true cost-of-production and that the marginal generator determines the system-clearing price at the given level of demand. If a company decides to manipulate the market, it will withhold offering into the market the next economic generator, which would potentially be the marginal unit. By the same token, a generator that could be a price-taker at its lower operational price could be withheld. Under both these scenarios the market price will be higher than with the withheld generator in service. Clearly with this approach, all generators already committed to the market will receive a higher price, including those generators that are selected by the market from the company which withheld its last economic generator. The company will still experience a higher profit, if the opportunity-cost given up by the withheld generator is lower than the increased profits by the generators already in service by the same company. As an example, assume that a company owns 3 generators, each having capacity of 100MW each, with incremental cost rates of 1.2 $/MWh, 2.0$/MWh and 2.6$/MWh, respectively. Also assume that there are 7 other generators, each owned by independent companies and that each of these generators has 100 MW in capacity, with each having incremental cost rates of 2.5 $/MWh. Based on a strict incremental-cost-rate merit order, 2 of the generators owned by the single company would be selected for dispatch first, while the balance will be supplied by any other

generator. If the load to be served is 400 MW, the corresponding static *HHI* value will be $(30)^2+ 7x(10)^2 = 1600$ while the *DHHI* will be $16.67^2+5x16.67^2 = 1666.7$. However, if the company withholds its 2nd generator, the next economic generator will be chosen. Under this scenario, it is clear that the *HHI* does not change, since this index uses the static installed capacity of all the generators. However the corresponding *DHHI* would reflect the change in the market share for the company withholding. The resulting *DHHI* for this case is $33.3^2+4x16.67^2=2222.2$. Under the circumstance since there would be more capacity available from the generator that did not participate in the market, a higher *DHHI* for the market will be realized. This in turn will send the correct signal to the market operator, indicating that the company is resorting to market manipulation practices.

The second approach companies resort to in manipulating the market consists of raising the price at which a given generator is offered into the market. In this scenario, there are three possible outcomes. The first occurs when the generator's offer-price is above the market-clearing price. Since there are many other lower priced generators available, the generator which inflated its price offer will not be chosen. It is clear that under this condition the market manipulation strategy of the company fails. The second possible outcome occurs when the generator's offer price is below the market-clearing price. Here the generator will become a price-taker and hence will receive the market-clearing price and appear to have a lower profit margin, without affecting the overall market. In the third possible outcome, the gaming company becomes the marginal unit; it would then reap the profits yielded by the inflated price but would have to be careful to ensure that its price is still below the next most economical generator. The net gain from gaming would then depend on two further factors; the predictability of remaining as the marginal generator, as well as the market-share and price-sensitivity of the marginal generator.

Looking further at the practices adopted and recommended by the *FERC*, it is evident that it relies heavily on the static *HHI* when investor-owned utilities apply for merger approvals. The use of this index and the procedures governing the process is elaborated in its order 592-policy statement [32]. By the same token, *FERC* has also urged each of the regional transmission organizations to adopt more timely and accurate automatic mitigation strategies than the current time-lagged mitigation processes used. That the current mitigation practices are performed after-

the-fact, causes serious problems for the *RTO*'s in justifying their mitigation actions, which entail corrections and adjustments to invoicing statements. As is evident, the current approach creates a large number of disputes that need to be resolved, and this requires time and effort from each of the parties involved. A better approach would be to conduct the evaluation of market power at the same time the market is cleared so that the dispatch decisions consider the effects of market concentration/ market-power, thereby avoiding any after-the-fact changes to the invoice statements.

This dissertation therefore attempts to formulate a new methodology to manage the market-power of participants while operating the day-ahead market at the lowest operational cost. Since *FERC* is well versed with the static *HHI,* an index that closely follows it but has alleviated its deficiencies, namely the *DHHI,* was selected for this study. As will be demonstrated, the opportunities presented to generator owners participating in the market are better captured in the *DHHI*. A look at issues California ISO, a regional transmission organization, faced during the 2000-2001 period clearly demonstrates the deficiencies of the static *HHI*, since the values reported were very low, although widespread gaming had taken place. This demonstrates that the static *HHI* for market concentration evaluations is deficient and hence justifies looking for an alternative but similar index whose evaluations take account of the true opportunities available for gaming.

In the day-ahead market, the market-power indices for each market hour need to be calculated. One way to verify that the market-power of all companies is in check for the entire 24-hour period would be to find the average modified *DHHI* over an entire day. Calculation of average *DHHI* (*ADHHI*) can be represented mathematically as follows:

$$ADHHI = (\sum_{j=1}^{24} \sum_{i=1}^{N(D)} [s_i(D(j)^+)]^2) / 24 .$$

In the evaluation process, at each economic dispatch strategy, the resulting $s_i(D(j)^+)$ will vary. Here, only those units that can still offer into the market will be used in deriving the

DHHI. In the strategy I am proposing, an acceptable dispatch solution will be obtained while keeping the value of *DHHI* below industry recommended values.

CHAPTER 5 - Survey of Prospective Optimization Algorithms for Market Clearing Problem

In using the mathematical representation of the day-ahead market dispatch to control market manipulations by the participants, one needs to find a solution algorithm that allows for multi-objective optimization. The selection of a suitable method becomes even more difficult when the solution space is known to have local minima and the solution space is multi-dimensional. While traditional algorithms such as linear programming, e.g., in the work of Madrigal and Quintana [23], have been used in the past to solve market-clearing problems, they predominantly suffer from having to search the entire spectrum to find the optimal solution. These traditional methods therefore make the time and effort spent on finding the solution unacceptably long and also suffer from getting trapped near local minima from time to time, thereby preventing the iterations from getting to the desired global minimum. When dealing with large power systems where there are thousands of generators and loads in the parameter space, the solution process becomes cumbersome if all possible combinations are to be evaluated. Due to these shortfalls in the traditional solution algorithms, researchers have diverted their attention to alternative optimization techniques.

Furthermore, most real-life problem spaces include minimization of multiple competing objectives. Most traditional solution techniques either force the solutions to combine multi-objectives into a single objective-function, or they solve each separate objective-function one at a time, as is done in classic primal-dual solution algorithms such as LaGrangian Relaxation (*LR*). As explained in the work of Dekrajangpetch and Sheble [13], in using a *LR* algorithm for a power auction implementation, the main mathematical problem is broken into two: a primal objective-function and a dual objective-function. The quality of the solutions obtained by this algorithm is represented by what is known as the "duality gap" or the spread between the primal and dual objective-function values. The larger the gap between the two functions the more uncertain one would be of the quality of results. The solution of the problem becomes even

harder when there are constraints that cannot be violated. Due to such deficiencies, researchers have been forced to look for newer solution techniques.

In recent years global optimization algorithms imitating or borrowed from certain principles of nature have proven their usefulness in various applications. For example, algorithms based on annealing processes, algorithms based on the central nervous system, cognitive human learning algorithms, and algorithms based on biological evolution have become popular. Of these, simulated annealing algorithms (*SA*), which are based on physical phenomena, tabu search algorithms (*TS*) based on human learning phenomena, and the genetic algorithms (*GA*) based on biological phenomena have gained popularity over the years due to their optimizing merits. The basis of some of these algorithms is biological, while others imitate social processes in their formulation.

5.1 Single and Multi-Objective Optimization

Most of the traditional algorithms reformulate a given multi-objective optimization problem into a single objective-function to be minimized or maximized. As an example, the revised simplex linear programming algorithm used by Huang and Song [35] to solve the constrained-power-economic-dispatch –control-problem is a good illustration of how a single objective-function is formed by reducing the given problem to a representative cost function. In this method a single objective-function is formed by combining the multi-objectives to be optimized by assigning relative weights to represent the importance of each of them. In order that this solution approach works, an *a priori* assumption of the relative importance of each objective has to be incorporated. This forces the solution to be guided in a given direction based on the judgment of the investigator. In order to prevent this subjectivity coming into the solution space, the concept of *pareto* dominance has been introduced. According to this principle, instead of giving an absolute (scalar) value to a solution, a partial order is defined based on *dominance*. A solution is said to *dominate* another solution when it is better on one objective, and not worse on all the other objectives. Considering a decision parameter vector x in a parameter space X, a decision vector $a \in$ X is said to dominate decision vector $b \in$ X if and only if,

$$\forall_{j \neq i,} O_j(a) \leq O_j(b) \text{ and } O_i(a) < O_i(b) .$$

This assumes without loss of generality that the objective functions $O_1,\ldots,O_n$ need to be optimized on all independent vectors *a* and *b* considered in the mathematical model. An objective is said to be *non-dominated* if no solution can be found that dominates it. The definition of the dominance relation gives rise to the definition of the *pareto-optimal-set*, also called the *set of non-dominated solutions*. This set contains all solutions that balance the objectives in a unique and optimal way. Since there is no single scalar judgment, this set usually contains a wealth of solutions. As there is no notion present of one objective being more important than another, the aim of multi-objective optimization is to provide this entire set. Picking a single solution from this set is then an *a posteriori* judgment, which can be done in terms of concrete solutions with concrete trade-offs, rather than using predetermined weighting of objectives.

5.2 Tabu-Search Algorithm

Recognizing the strengths of cognitive learning, and also the deficiencies of other algorithms such as artificial neural networks and simulated annealing, an algorithm that is based on this cognitive-learning principle is worth investigation. Of the algorithms developed, the tabu-search-meta-heuristic algorithm shows strong potential. Since its introduction to the scientific world, many researchers have made a number of contributions to enhance its features and capabilities. The word “tabu” or “taboo” comes from the language Tongan, used by aborigines in the island of Tonga, to indicate things that cannot be touched because they are sacred. According to the Webster Dictionary [50], the word tabu means, “a prohibition imposed by social custom as a protective measure” or “something banned as constituting a risk”. Hence, the algorithm follows the basic premise of avoiding counter-productive courses and retaining memory of those unsuccessful attempts while moving the overall solution of the given problem towards its global minimum. However, in the process, one cannot overlook the important association with the traditional usage where the tabu gets conditionally modified over time, based on the circumstances and events that succeed the initial tabu imposition. Therefore, to qualify a tabu search algorithm as intelligent, one has to consider its key features of adaptive memory and responsive exploration. The adaptive memory feature brings about effective and economic searching of the solution space. Due to the fact that the local choices are guided by information

collected during a search, tabu-search contrasts with memoryless designs that heavily rely on semi-random processes based on sampling. Examples of such memoryless algorithms include common semi-greedy heuristics, and annealing processes that heavily rely on the laws of physics. The emphasis on collective memory in tabu search derives from the basic premise that a bad strategic choice can yield more information than a good random choice. In a system that utilizes memory, a bad choice based on strategy can provide more useful clues about how the strategy may be profitably changed.

Responsive exploration on the other hand utilizes the basic principles of intelligent search in which good solution features are exploited by exploring promising neighborhoods near the good solutions found. In order to effectively design a good tabu algorithm, key memory structures that retain recentness, frequency, quality and influence must be included. As one can see, recentness and frequency complement one another. The quality dimension refers to the ability to differentiate the merits of solutions visited during a given search. Memory can be used to identify elements that are contributing to good solutions and the paths that lead to such good solutions. The next dimension, "influence", considers the impact of the choices made during a search, not only on quality but also on the structure of possible solutions. Recording information about the influences of choices on particular solution elements incorporates an additional level of learning. A good tabu search algorithm utilizes multi-faceted memory structures and flexibility to allow the search to be guided in a multi-objective environment.

The solution approaches used in tabu search meta-heuristic can be characterized as identifying a neighborhood of a given solution, which contains other so-called transformed solutions that can be reached in a single iteration. A transition from a feasible solution to a transformed feasible solution is referred to as a *move*. A starting point for tabu search is to note that such a move may be described by a set of one or more attributes, and these attributes when properly chosen can become the foundation for creating an attribute-based memory. Following a steepest descent / mildest ascent approach, a move may either result in a best-possible improvement or a least-possible deterioration of the objective-function value. Without additional control, however, such a process can cause a locally optimal solution to be re-visited immediately after moving to a neighbor, or in a future stage of the search process.

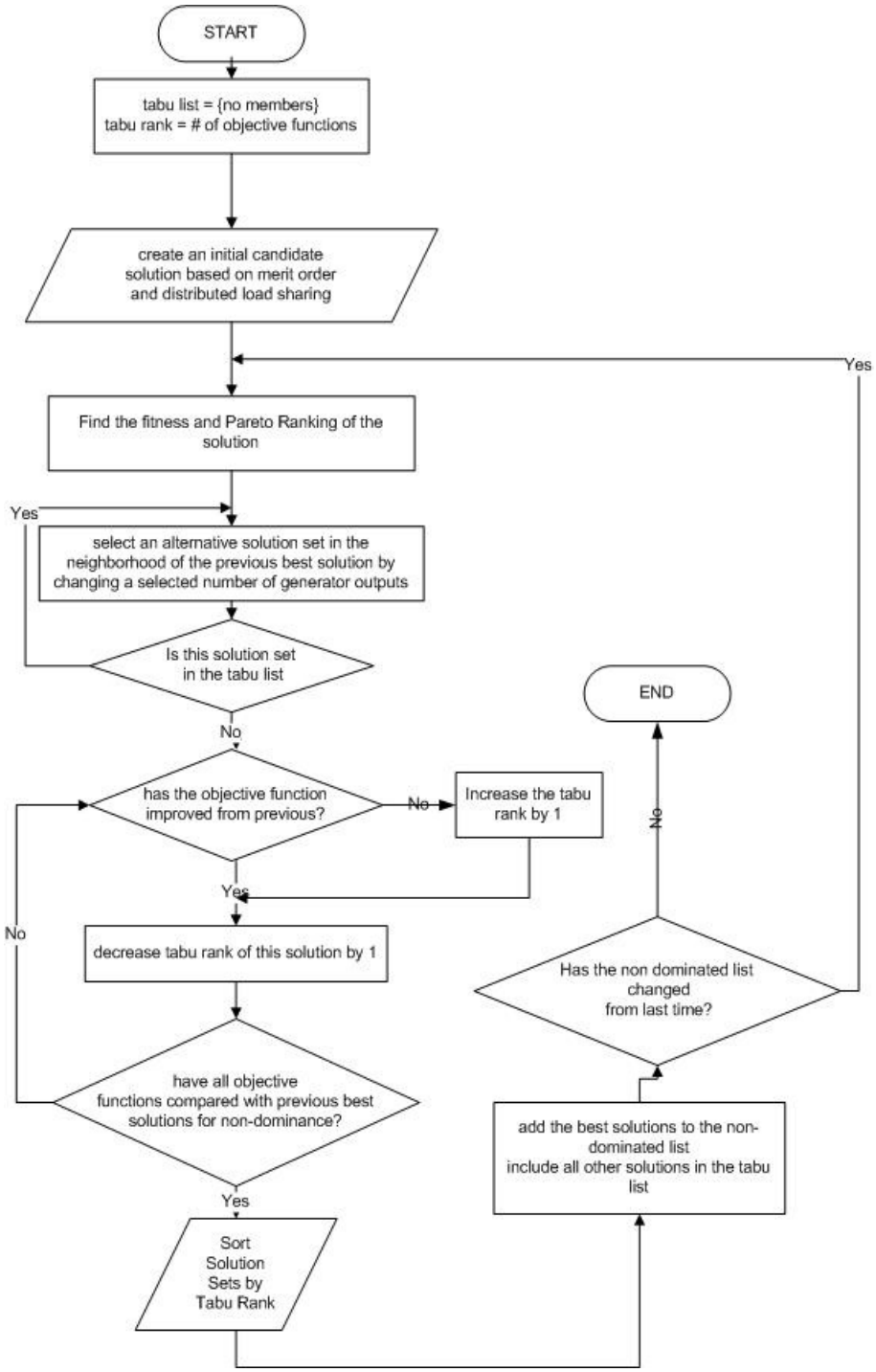
To prevent the search from endlessly cycling between the same solutions, the attribute-based memory of tabu search is structured at its first level to provide a short-term memory function, which may be visualized to operate as follows. Imagine that the attributes of all explored moves are stored in a list, named a *running list*, representing all solutions previously encountered. Then, related to a sub-list of the running list a so-called *tabu list* may be introduced. Based on certain restrictions, the tabu list implicitly keeps track of moves or more precisely, salient features of these moves by recording attributes complementary to those of the running list. These attributes will be forbidden from being embodied in moves selected in subsequent iterations because their inclusion might lead back to a previously visited solution. Thus, the tabu list restricts the search to a subset of admissible moves consisting of admissible attributes or combinations of attributes. The goal is to permit "good" moves without re-visiting solutions already encountered from one-iteration to the next.

5.3 Multi-objective Tabu-Search-Meta-Heuristic Algorithm

Given that many real-life problems have multiple objectives to be optimized, the next focus should be evaluating the potential of the tabu-search algorithm to be used for multi-objective optimization. Many researchers recently have extended the traditional tabu search principles to solve multi-objective problems [25], [27], [28]. The multi-objective tabu search algorithm in this research uses the principles of tabu ranking and tabu list approaches in its solution criteria. Here, the solutions based on a neighborhood, tabu list and a tabu ranking list are compared with one another. The solutions that have similar tabu-ranks are grouped together. Those solutions that have the lowest *tabu ranks* are incorporated into the tabu list and visiting them will be prevented in subsequent evaluations. Prospective solutions from one evaluation cycle to the next are drawn from solutions that are in the neighborhood of previous solutions. These selections will always be compared with the tabu list to ensure none is on the list.

The general outline of the multi-objective tabu search procedure can be represented by the flow chart shown in figure 5.1.

Figure 5.1 - Flow Chart of the Multi-Objective Tabu Search Algorithm



5.4 Genetic Algorithms

Genetic algorithms are inspired by Darwin's theory of evolution [20], [21], [22]. In the theory of genetics, a cell is considered as the basic building block that constructs every living organism. However, each cell in turn has constituent building blocks called *chromosomes*. Chromosomes in turn are made up of strings of DNA that serve as models for each organism. When a block of DNA is combined together it is called a *gene*. Each gene encodes a particular protein or a trait. For example the color of eyes of every human being is a *trait*. Possible value settings in a gene to bring up a given trait such as blue or brown in eyes are called *alleles*. Each gene has its own position in the chromosome. This position is called a *locus*. A complete set of genetic material for all chromosomes in an organism is called a *genome*. A particular set of genes in a genome is called a *genotype*. The genotype with later development after birth forms the base for the organism's *phenotype*, its physical and mental characteristics, such as eye color, intelligence, etc.

Birth of offspring from parents is known as the process of reproduction. During reproduction, essential traits of each parent are carried forward into the child's chromosomes. This process is identified as *recombination* or *crossover*. Here, genes from parents form into whole new chromosomes. These new chromosomes then go through what is identified as *mutation*. Mutation is the process whereby elements of DNA are slightly modified. Modification is actually caused by errors in copying from parents in the reproduction process. This however, causes offspring to show their own unique characteristics even though exhibiting traits from both parents they originated from. Once an offspring is created, its ability to reproduce will determine its success in maintaining its presence in future generations.

The popular genetic-algorithm optimization techniques use the basic biological principles described above. In these algorithms an approach similar to the evolutionary process is used. The solution process begins with a set of chromosomes identified as the parents that belong to a population. Chromosomes that are considered as the "healthiest" in a given generation are used to form the chromosomes of the new population in the next generation. The selection of the next population is motivated by the hope it will be better than the old one. The members of the subsequent generation are called *offspring*. All offspring are selected according to their fitness or

their ability to reproduce. The process of parents creating their offspring is repeated for a number of generations until some condition such as the total number of generations the process is allowed to go through or the relative distinction between a set of parents and their offspring becomes very insignificant. When a genetic algorithm is used for solving a given problem a number of factors have to be considered. First, key features of modeling a chromosome have to be identified. Then the basis of generating offspring chromosomes from the each of the parent chromosomes has to be determined. Here, heredity of each parent will be used for formulating the offspring. The notion of using heredity is defined by two basic operations: mutation and crossover. In the biology analogy, each chromosome of a given species can be represented by an encoding methodology. Use of binary encoding to represent chromosomes is a very common practice, although it is not suitable for solving every practical problem. Looking closely at the day-ahead market and its operation, one could clearly see the benefits of using a value-encoding methodology. Here the distinction between the value-encoding method and the permutation-encoding method has to be clearly identified. In contrast to permutation-encoding, where the order of selection of genes in a chromosome is determined by the problem, value-encoding uses values that represent a feature of the problem to be solved in the values in the alleles. An example shown in Figure 5.2 best illustrates the principles of encoding. Considering the application of a day-ahead market dispatch problem, each allele value in a given chromosome represents the generator output needed to supply a given load in the system. The values in each allele in each parent depicted in Figure 5.2, show the order in which generators are chosen in supplying the given system load for a given market hour along with corresponding output from each generator. Based on this approach, the values 30, 45, 25, 45, etc., depicted in parent 1, indicate that when supplying the system load, the first generator will supply 30 MW, while the second generator will supply 45 MW. The generation offered by the next 2 generators under this arrangement will be 25 MW and 45 MW, respectively. This process of constructing the chromosomes will be continued until the total system load is met. The generation supply arrangement with the second parent chromosome will be in the order 45, 30, 70, etc. Using each of these parent chromosomes the system load of 260 MW will be met. However, it is worth noting here that in order for a system to be able to meet the total load demanded, the total generation available must exceed the highest load that needs to be supplied at any hour of the day. As an example, if the peak-load to be supplied for a given market day is 350MW, the total

generator capacity available to the market must exceed this 350MW, although the load demanded at different hours of the day would be less than this peak load.

The principles of crossover are next used to create an offspring from its parents. In the process, selected genes from one sub set of parents are chosen and mixed with different genes from other parents. This process is identified as crossover is best explained by an example as depicted in Figure 5.3. The example described here, represents a very basic crossover method in which a single crossover point was selected. Depending on the nature of the species and the stage of evolution, the crossover principles can take many different forms. In the method considered here, genes that are to the left of the crossover point from the first parent are combined with genes to the right of the crossover point from the second parent to form an offspring. Thus a new generator order is created in forming the offspring. To avoid duplication of generators in a chromosome due to crossover, only those pairs of parents who have the same generators to the right of the crossover point are used in the crossover process. Any pair of parent chromosomes that do not satisfy this condition will not be chosen for the crossover process. An example using 3 parent chromosomes would elaborate the selection process described above. Let us assume that a 1^{st} parent chromosome has a generator order of 2,6,4,1,7,8,5,3, while a 2^{nd} parent chromosome has a generator order of 8,1,5,7,2,3,6,7 and a 3^{rd} parent chromosome in the population has a generator order of 1,2,4,6,3,5,8,7. Considering the first two chromosomes, a crossover point that allows the exchange of generator order between these two chromosomes cannot be found and therefore the second chromosome will be discarded from the viable list of parent chromosomes for crossover with the first parent. On the other hand, considering the 1^{st} and 3^{rd} chromosomes, a crossover point after the 4^{th} gene would allow these two chromosomes to cross over easily and will form a viable pair. It is worthwhile to note that based on the crossover methodology adopted for solving the day-ahead market problem, formation of parent chromosomes that are unable to cross over with one another will be avoided, as explained in the next chapter. The associated generator outputs are then assigned to the offspring chromosome, so that the total generation is met by the arrangement. Once the crossover process is complete, the child's chromosome undergoes a process called mutation. Mutation is the process leading an offspring to have its own identity. An example of a simple mutation process is demonstrated in Figure 5.4. Here mutant offspring are formed by replacing two or more alleles from the original offspring. In the example

presented below, the allele 2 and 4 of the original offspring 1 and original offspring 2 have been first exchanged and then replaced with two new values to form two new mutant offsprings. This process creates alleles in the mutant offspring that are not found in the parent chromosomes. The fact that the mutant offspring have their unique features deviant from their parents is demonstrated here. Similar to the crossover process, the assignment of gene values that represent the output from each generator are adjusted so that the total load to be served is met. Using the example depicted in Figure 5.4, since all other allele values are kept constant from the parent chromosomes, in order to retain the total system generation at 260MW, the two mutated allele values in offspring 1 have to sum up to 90 MW, while the mutated allele values in the offspring 2 have to sum up to 75MW. In the two examples mutated values of 60 and 30 for offspring 1 and 40 and 35 for offspring 2, which are not values found in the parent chromosomes, could be chosen. These are selected at random while ensuring that the total combined outputs from the 2 alleles are kept at the required 90MW and 75MW, respectively. As depicted in the example below in the mutation process, not only will the order of the generators be changed, but also allele values will be changed from the parent chromosomes. Also, to keep chromosomes viable for crossover as explained on the previous page, only the genes to the left of the crossover point will be selected for mutation.

Figure 5.2 - Parent Chromosomes

Parent 1	30 45 25 45 60 15 30 10
Parent 2	45 30 70 45 15 10 20 25

Figure 5.3 - Offspring Chromosome

Parent 1	30 45 25 45 60 | 15 30 10
Parent 2	45 30 70 45 15 | 10 20 25
Offspring 1	30 45 25 45 60 | 10 20 25
Offspring 2	45 30 70 45 15 | 15 30 10

Figure 5.4 - Mutated Offspring

Original offspring 1	30 45 25 45 60 | 10 20 25
Original offspring 2	45 30 70 45 15 | 15 30 10
Mutant offspring 1	30 60 25 30 60 |10 35 25
Mutant offspring 2	45 40 70 35 15 | 15 30 10

The selection of a crossover and a mutation mechanism as demonstrated in the examples in the previous section will not alone guarantee that the generations will migrate toward forming healthy offspring. The level of crossover and mutation applied when forming a new generation from a parent generation governs the success of the evolution process. Typically a predetermined percentage of parent chromosomes from the total population are selected for crossover at every generation. In theory one could choose a crossover percentage between 0% and 100%. However, given that at 0% crossover the new population will be an exact copy of the previous generation, selection of this crossover level is generally avoided. On the other hand if the new population is formulated using a 100% crossover, all offspring formed will be completely different from the parents. This would eliminate all good traits of the parent chromosomes from the offspring and is not recommended due to the fact that there will be no assurance of convergence. Many practical applications of genetic algorithms have used crossover percentages around 95% with the intention of allowing some of the fit parent chromosomes to migrate to the next generation without any alterations.

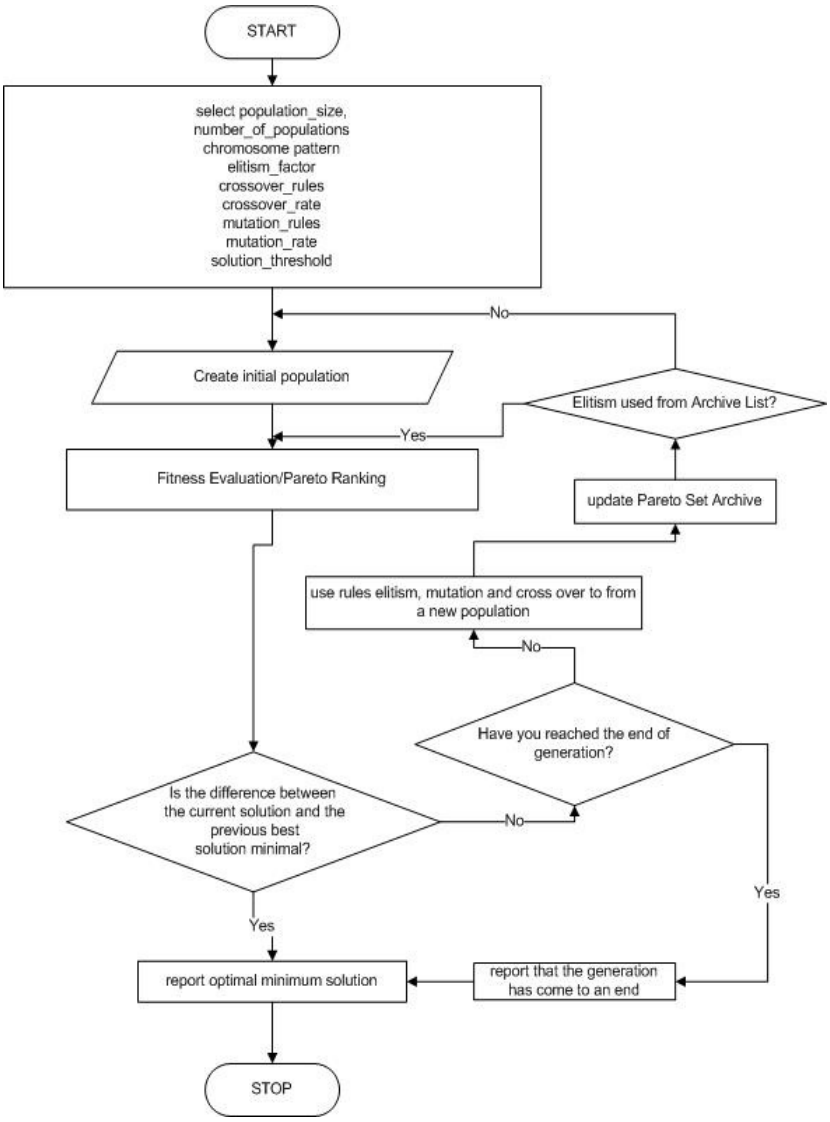
Typically a very low mutation rate is selected to reduce the amount of randomness introduced into the solution. Selecting an appropriate level of mutation is the key to preventing a genetic algorithm from getting entrapped in local minima.

Before proceeding with selecting a crossover and a mutation scheme, a given genetic algorithm must first use a solid method of forming new offspring from its parent generation. There are many different techniques that are commonly used for accomplishing this. Following

is a listing of some of the most commonly used methods. Probing further one can see that some of these methods are mutually exclusive of others, while some methods can be used in combination. One selection process is known as *elitist selection*. Here, best members from each parent generation are selected and retained to be included when forming the next generation. Experience shows that using pure elitism should be discouraged. Many studies recommend the use a modified form of *elitism,* in which only the single best or a few of the best individuals from each generation are retained. A second popular approach is known as *Roulette-wheel* selection. This method is based on rating the chance of one individual being selected over its competitor. Conceptually, as the name implies, this selection process is very much akin to a game of roulette where each individual gets a slice of the wheel. The size of the slice assigned from the wheel will be dependent on the fitness of a given chromosome. In the method, fit offspring get a larger slice of the wheel assigned to them, while the less fit ones get smaller sizes assigned. The wheel is then spun, and whichever individual "owns" the section on which the pointer lands each time is chosen. The concepts of a roulette wheel is easily modeled using number ranges to represent the slice of the wheel, while a random number generator could be used for representing the spinning of the pointer. Another method commonly known as *scaling selection* uses a criterion based on the strength of the selective pressure. In this method the probability of selection increases as the average fitness of the population increases. This method is more appropriate when all candidates have a relatively high fitness rating and a small difference distinguishing one another. Another commonly used selection criterion is *tournament selection*. This method selects multiple subgroups from the original population and the reproduction is limited to parents from these subgroups. A member from one subgroup is combined with another member from a different subgroup to form an offspring. *Hierarchical selection* is yet another common method. According to the principles of this method, individuals go through multiple rounds of selection in each generation. Lower-level evaluations are faster and less discriminating, while those individuals that survive to higher levels are evaluated more rigorously. The advantage of this method is that it reduces overall computation time by using faster, less selective evaluation to weed out the majority of individuals that show little or no promise only subjecting those who survive this initial test to more rigorous and more computationally expensive fitness evaluation.

Population size also plays a vital role in a genetic algorithm (*GA*). Population size in this context does not represent all possible solution permutations to a given problem. The population here represents a sample that is chosen to be representative of the entire solution set. Typically the population size of a *GA* is kept at a fraction of the entire solution set. The number of chromosomes in a generation will govern the time for finding an optimal solution to a given problem. If there are too few chromosomes, *GA* has few possibilities to perform crossover and only a small part of the search space is explored. This may result in *GA* ending up with a sub-optimal solution. On the other hand, if there are too many chromosomes, *GA* will slow down, outweighing the attractiveness of this algorithm over the traditional solution techniques. Research shows that moderate-sized populations are best suited for many practical problems. A flow chart for a typical *GA* solution algorithm is represented in Figure 5.5.

Figure 5.5 - Flow Chart representing a Genetic Algorithm



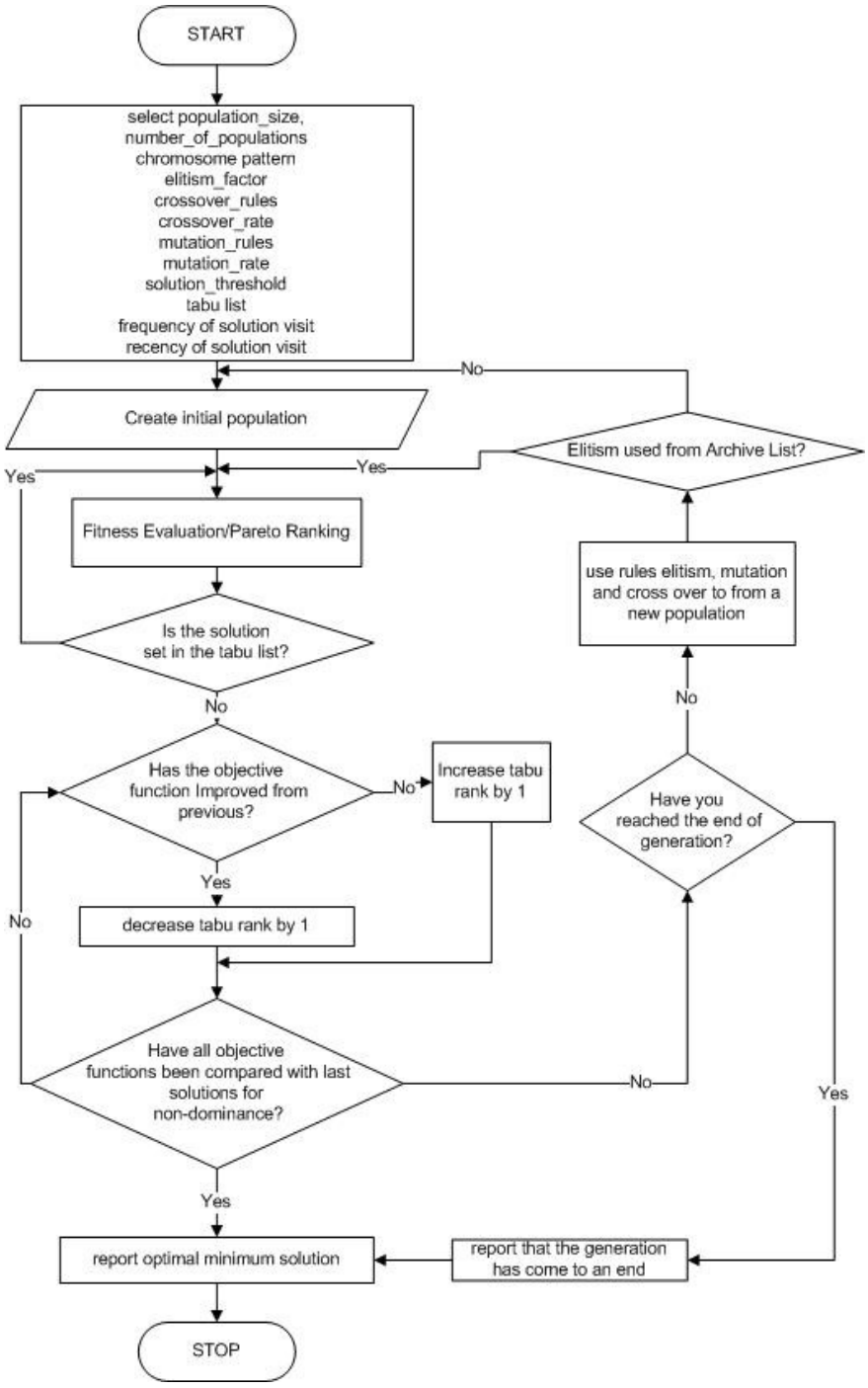
5.5 Combined Multi-Objective Tabu/Genetic Algorithm

In the combined tabu/genetic algorithm solution methods the advantages of both systems are exploited while neutralizing some of the deficiencies of each of the base algorithms. The tabu search algorithm tends to look around the neighborhood of a selected solution. While it is capable of moving towards a solution fast once in the neighborhood, depending on the initial

selection, it may not be able to converge to a global minimum. Since on many occasions the initial solution is selected at random, the chance of selecting a bad initial solution is high. If this happens the algorithm will not be capable of converging to the global optimum solution since this algorithm concentrates on searching solutions in a predetermined neighborhood. On the other hand, retaining memory of bad neighborhoods visited during the attempts to arrive at a solution and avoiding these bad neighborhoods is a key merit of the tabu search algorithm. Conversely, the solution quality from a genetic algorithm tends to suffer when the solution space becomes large. However, a genetic algorithm has the ability to identify fit solutions and generate offspring without being limited to a fixed neighborhood. Since traditional genetic algorithms do not have any mechanisms for retaining their previous actions, an algorithm that combines the cognitive learning capabilities of a tabu search algorithm with a genetic algorithm would offer great potential. This combined approach can be justified through a real life example where some plants are cross-pollinated to form strong offspring, while based on previous experiences some species are not allowed to cross-pollinate since the resulting offspring would be too weak to survive. The operational logic of the combined Tabu/GA algorithm is presented in Figure 5.6. As described in the flow chart, the initial population in this algorithm will be formulated similar to that in a genetic algorithm. However, the successive population selections will be done in a given neighborhood as in the Tabu algorithm, and a tabu list will be maintained to ensure that previously visited solutions that have been "tabooed" are not visited in subsequent populations.

Previous work done using a combined Tabu/GA algorithm [16] shows that the results obtained were much better than those obtained using a plain GA algorithm. As represented in [17] the combined algorithm is found to be faster and more efficient than each algorithm used in isolation. The results are depicted for problems with non-linear and discontinuous objective functions. The work other researchers have done to solve engineering problems and their reported successes made this algorithm stand out as a promising one to be investigated.

Figure 5.6 - Flow chart of the combined Tabu/Genetic Algorithm



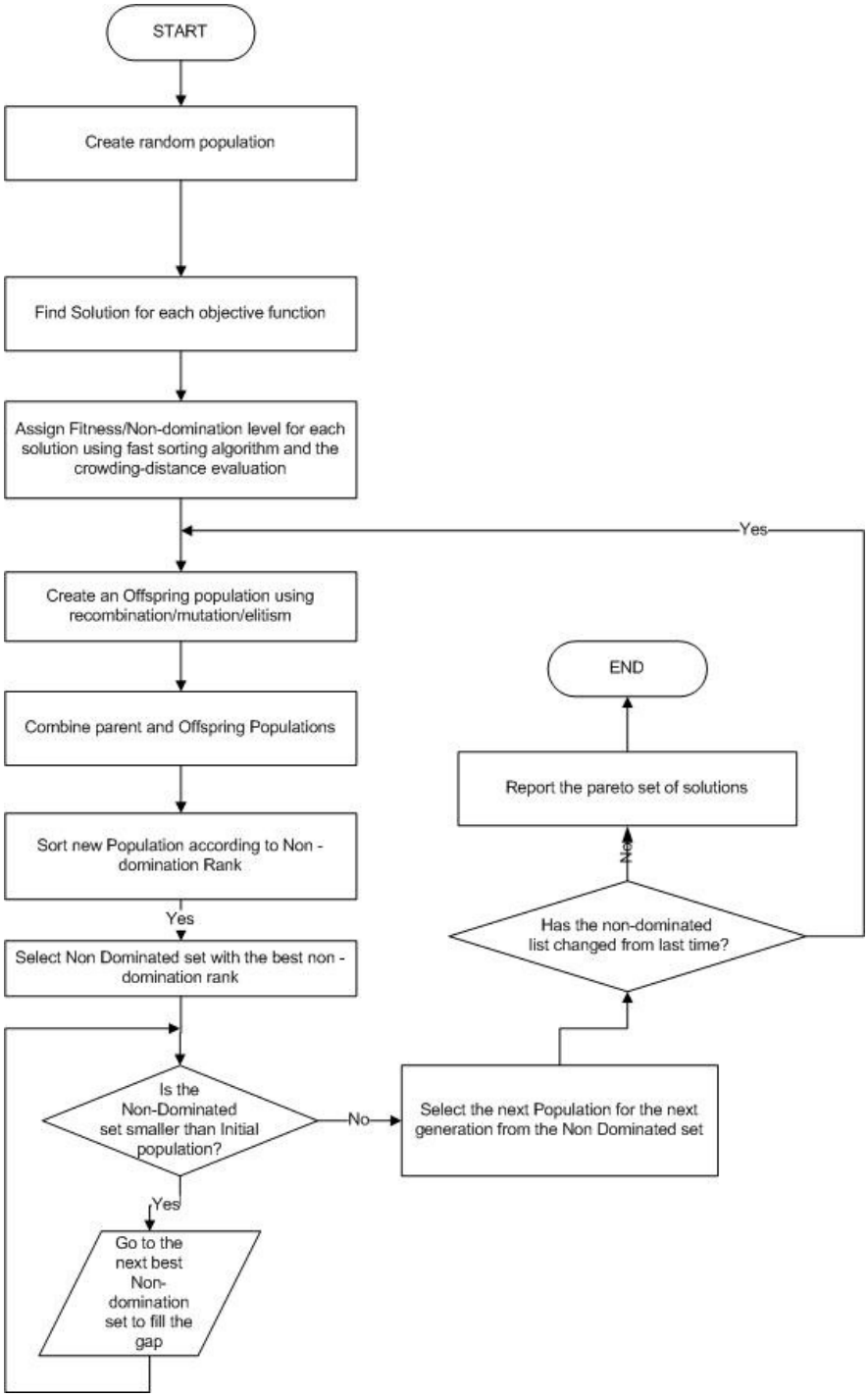
5.6 Non-Dominated-Sorting-Genetic Algorithm: *NSGA-II*

As described in the previous sections, the presence of multiple objectives gives rise to a set of optimal solutions, commonly known as *pareto-optimal* solutions, rather than a single optimal solution. Classical optimization methods suggest converting the multi-objective optimization problem to a single-objective optimization problem by selecting one *pareto-optimal* solution at a time. This necessitates re-simulating for every alternative condition. In order to alleviate this need for repeated simulations the multi-objective evolutionary algorithms have become increasingly popular in the recent years. Using a multi-objective evolutionary algorithm, multiple optimal solutions can be achieved in a single simulation run. In these approaches, a simple evolutionary algorithm (*EA)* is extended to maintain a diverse set of solutions with the emphasis on moving toward a true *pareto-optimal* region. The nondominated sorting genetic algorithm proposed in [36] was one of the first such algorithms. Over the years many researchers have pointed out some deficiencies of this algorithm. High computational complexity of the nondominated sorting method used, failure to use an *elitism* scheme in the solution algorithm, and the requirement for specifying a sharing parameter in the algorithm are some of those identified. Based on these issues, Deb, *et al.* [19], have proposed an improved version of their *NSGA* algorithm called *NSGA II*. In their paper [19] a comparison of two other powerful algorithms, Pareto-archived evolution strategy (*PAES*) and *strength pareto EA* (*SPEA*) with *NSGA II* algorithm for different types of problems, shows that this algorithm out performs its competitors when used for solving widely varying problems. One major advantage an algorithm might possess is the ability to include constraints in the solution space. *NSGA II* algorithm has the potential for handling constrained problems with ease. This makes the *NSGA II* algorithm much more suitable for real life problems such as the day-ahead dispatch problem being considered in this research.

Two distinct entities are calculated in *NSGA II* to validate the quality of a given solution. The first is a domination-count where the number of solutions that dominate a given solution is tracked. The second keeps track of how many sets of solutions a given solution dominates. In the

process, all solutions in the first non-dominated front will have their domination count set to zero. The next step is to select each solution in which the non-domination count is set to zero and visit all other solutions in the solution set and reduce the domination count by one. In doing so, if the domination count of any other solution becomes zero, this solution is grouped in a separate list. This list is flagged as the second non-dominated front. This process is then continued with each member of the second list until the next non-dominated front is identified. The process is continued until all fronts are identified. Based on the non-domination count given to a solution, a non-domination level will be assigned. Those solutions that have higher non-domination levels are flagged as non-optimal and will never be visited again. One of the key requirements of a successful solution method is ensuring that a good representative sample from all possible solutions is chosen. Introduction of a density estimation process and a crowded-comparison operator has helped *NSGA II* to address the above need. The crowding-distance computation requires sorting a given population according to each objective function value in an ascending order of magnitude. Once this is done, the two boundary solutions with the largest and smallest objective value are assigned distance values of infinity. All other solutions lying in between these two solutions are then assigned a distance value calculated by the *absolute normalized* distance between each pair of adjacent solutions. After each population member is assigned a crowding-distance value, a crowded-comparison operator is used for comparing each solution with the others. This operator considers two attributes associated with every solution, which are non-domination rank and crowding-distance. Every solution is rated with others based on the non-domination rank. Solutions with lower ranks are deemed better in this attribute. Once all solutions that belong to the best front are chosen based on the non-domination rank, the solution that is located in a lesser-crowded region is considered better and forms the basis of the *NSGA II* algorithm. The flow chart depicting the *NSGA II* algorithm is shown in Figure 5.7.

Figure 5.7 - Flow chart of the *NSGA II* Algorithm



CHAPTER 6 - Application of Evolutionary Algorithms to the Day-Ahead Market Problem

This chapter presents the criteria used in formulation of the day-ahead market problem using three of the most promising solution algorithms. Since the solution of a given problem requires careful modeling to fit into the chosen solution algorithm, the preliminary sections will attempt to present the rationale used in the modeling process.

6.1 Multi-Objective Tabu-Search Algorithm

All problem-specific parameters were chosen as described below to model the given day-ahead market dispatch problem. The assumptions and rationale for selecting a given value for each parameter are described in detail in the following sections.

6.1.1 Initial Solution Selection

The convergence to the optimal solution in the tabu-search algorithm is greatly dependent on selecting a suitable initial solution. Given that the tabu-search algorithm relies on searching only in a given neighborhood, time and effort spent on selecting the starting solutions is worthwhile. Although an initial solution can be chosen at random, an initial solution that is merit-order based on price characteristics of the generators, combined with the corresponding market-power indices was chosen in this study. First, all generators were ranked based on their individual incremental price curves. Next the relative market-power index for each generator at the selected merit-order commitment was calculated. This selection approach is best explained by an example. Consider that a given power market had 3 generators available for supplying a day-ahead market, where the incremental prices for three generators are 2.0, 1.2 and 2.5 $/MW, respectively. Moreover, these generators are capable of supplying up to a maximum of 100MW, 30MW and 150MW, respectively, to the market. If the generators are selected at random,

generator 2 could be selected first, followed by generator 3 and generator 1 to supply the system load for a given market hour. Assuming that the system load is 225 MW, generator 2 would supply 30MW, followed by generator 3 and generator 1, supplying 150MW, 45MW, respectively, to meet the total system load. The corresponding *DHHI* for this arrangement is 10000, since only generator 1 has capacity left to offer to the market. The cost of supplying the market with the above dispatch arrangement is $501.00. As an alternative, if a merit-order based on incremental costs of generators was used, generator 2 will be chosen first, followed by generators 1 and 3. Even under this arrangement, if all economic generators are allowed to supply the market up to their maximum capacity, it would still result in an overall *DHHI* of 10000, with an overall cost of operating the market of $473.50. Given that this approach is not any better than the random selection approach described previously, a second selection criterion to arrive at an initial *DHHI* that is lower than the maximum value of 10000 was considered. In this process, the generator that is the most economical is operated to its capacity. The remaining system load is then supplied by generators 1 and 3, each operated at a level that is equally below its maximum capacity. For the example considered, with generator 2 operated at 30 MW, the remaining system load to be supplied becomes 195 MW. With a combined total maximum capacity of 250MW between generators 1 and 3 and a combined load of 195 MW to be supplied by these generators, a combined total of 55 MW of unused capacity will be available from these two generators.With the spare capacity equally divided amongst these generators, generator 1 would supply 72.5MW (100 – 55/2) of the load, while generator 3 would supply 122.5 MW (150 – 55/2). The resulting *DHHI* for this arrangement using the formula described in Chapter 3 (page 19) of this dissertation is {((100.0-72.5)(100)/((100.0-72.5)+(150.0-122.5)))^2 + 0+ ((150-122.5)(100)/((100.0-72.5)+(150.0-122.5)))^2} = 5000. The corresponding total cost of supplying the market under this dispatch scenario was computed and found to be $487.25. The cost although higher than the previous dispatch arrangement clearly forces the market concentration index to be half of what it was before. A pre-filtration process based on the approach described above was adopted when formulating the initial solution candidates. The same process is extended when there are more than three generators in a given system; dispatch all but the two most expensive generators to their economic maximum limits, while allowing the last two generators required to supply the remaining load equally, as described above. As an example, if we consider a power system with 10 generators and with a system load of 600MW, proceeding

with the approach described above, the most economical 8 generators will be dispatched in their merit order to their maximum default capacity; assuming that the total generation offered by these 8 generators is 500 MW, the remaining system load to be supplied by the most expensive generators would be 100 MW. If the remaining 2 generators had default maximum capacities of 100 MW and 150 MW, respectively, then the generator with a capacity of 100 MW will be operated at 25MW, while the second generator will be operated at 75MW to meet the required 100 MW of load. This way both generators are operating at 75 MW below their maximum capacity. This process of allocation will work for most generator pairs since the allocation process will always assign half of its total available capacity, except in the case of a very small generator which has a capacity that is less than half its share. As an example, if we consider the 9^{th} most expensive generator to have a capacity of 40 MW, considering the load of 100 MW to be supplied between the above generator and the 10^{th} generator with a capacity of 150 MW, the allocation for each generator would be (150+(40 -100))/2 = 90/2 = 45 MW. Since the 9^{th} generator only has 40 MW to offer, it would be unable to offer the 45 MW expected to be supplied under this scenario. When such a situation arises, the algorithm could be extended to include another generator, which would be the the next most expensive generator (or the 8^{th} most expensive generator in this case), to supply the uncommitted load with all other generators dispatched to their maximum limits in the merit-order. To elaborate this, when such a situation arises, the load to be supplied will be equally shared among the three most expensive generators instead of the two most expensive generators and each will be operated at a level that is one-third of the load to be supplied below its maximum capacity. The same process is repeated for all hours of the market day. The whole idea behind this approach is to use initial solutions that do not result in extreme values for the two objectives being optimized.

6.1.2 Selection of the Length of Tabu-List

The tabu-list refers to a running list of solutions that were previously found and are to be avoided as possible solutions in subsequent evaluations. If this list is too incomplete, the chances of revisiting an unsatisfactory solution will be high. This would bring about the possibility of iterations cycling without finding the optimal solution. If however, the list is made too large, it will too heavily restrict the solutions to be examined. The length of the tabu-list could be held

constant or could be varied from one iteration to the next. Based on previous work done by Glover and Anderson, who are considered pioneers of the tabu-search algorithm [37], the tabu-list length was held constant for the analysis conducted in this research. Many researchers who have used the tabu-search algorithm for practical applications have recommended the use of a tabu-list length between 7-15. A tabu-list length of 10 was used in the day-ahead market dispatch problem. With the tabu-list length constant, a running list of the 10 most recently visited solutions that are closest to the best solution but worse than the most recent best solution will be kept. Once a new solution is found, it will first be compared with the best solution found so far. If the new solution has improved, the last best solution will be added to the top of the tabu-list. This forces the 10^{th} solution in the tabu-list to roll off the tabu-list and each of the previous solutions to move down the list, making way for the newest addition. Also the most recently found solution will become the best solution to be improved. If on the other hand, the latest solution has not improved on the previous best solution, it will be added to the tabu-list at the appropriate list ranking. When the next set of solutions is selected, solutions in the tabu-list will be avoided since the list contains all those solutions most recently visited which are in the neighborhood of the best solution although not better than the best solution retained.

6.1.3 Neighborhood-Solution-Space

The maximum number of trial solutions considered in each iteration is referred to as the *neighborhood-solution-space-parameter* for a *TS* algorithm. Typically this value is set to be one less than the tabu-list length selected. Since the tabu-length was selected to be 10 when solving the day-ahead dispatch problem, a neighborhood-solution-space-parameter value of 9 was chosen based on the recommendations made by other researchers. The next step is to select candidate solutions for the neighborhood-solution-space. The best solution found through the last iteration cycle is used as the foundation for forming the candidate solutions in the neighborhood space. With this approach, possible candidate solutions in the neighborhood of pervious best solutions will be chosen by altering the generator output values for a selected subset of generators available for supplying the market. In the process of selection, potential candidate solutions are checked against the tabu-list to ensure that tabu solutions are not revisited. An example to elaborate the selection of a neighborhood solution space for a sample day-ahead market dispatch problem is described below. For a 3-generator dispatch problem which began

with a first solution of 30MW, 20MW and 30MW, a possible sample pair of neighborhood candidate solutions is, 25MW,25MW,30MW, and 30MW, 25MW, 35MW. In general, these candidate solution vectors are formulated based on the step by which each generator can move up or down from its current dispatch level ensuring that the generator maximum and minimum limits are maintained. In the example above all three generators are assumed to have adjustment steps of 5MW each. Hence an equal increase in one generator would be offset by an equal reduction in a second generator, so that the total power supplied by the generators meets the system load demanded. In order to truly be in the neighborhood of the first solution, a single generator will be first adjusted from its current operating point increasing its output based on its adjustment step. This increased output will then be compensated by either a single generator or a combination of generators in the pool by having their outputs reduced based on their adjustment steps, so that the total power generated is kept constant. As an example, if the adjustment step of the first generator selected at random to find an alternative solution in the neighborhood is 10MW, a second generator that has an incremental step of 10MW or 2 generators with incremental steps of 5MW each will have to be chosen to compensate for the adjustments made in the first generator so as to keep the total generation for the hour constant. Generally the number of generators adjusted in a single neighborhood search is maintained at around 2 to 3 generators at a time. If the system has more generators than is required to supply the total load demand in a given market hour, some will be turned off.

6.2 Multi-Objective Tabu/ Genetic Algorithm

In this method, the merits of multi-objective tabu-search algorithms and those of the genetic algorithms were combined. The approaches adopted when selecting the parameters for a multi-objective tabu-search algorithm are extended in this method so that a comparison between the *MOTS* algorithm and the *MOTS/GA* algorithm can be made. Selection of GA algorithm-specific parameters such as the encoding schema, crossover probability and mutation schemes, was done as described in the following sections.

6.2.1 Encoding Schema

A value-encoding schema was adopted to represent the day-ahead market dispatch in this research. This eliminated the need to define an additional translation schema to convert the generation offer-values into representative alleles at the time of building the chromosomes. The

length of a chromosome was chosen to be equal to the total number of generators that were participating in the market for a given hour for a given market day. For example, if 10 generators were participating in the market, then the length of the chromosome was chosen as10. The entries in the chromosome first represent the order in which the generators are committed, then corresponding power outputs of each generator for a given market hour are added to each allele value. A separate set of chromosomes for each hour of the market day with the corresponding entries representing the output of each generator at that hour is considered simultaneously. The order in which the generators are used for supplying the load for a given market hour is represented by the gene order in the chromosome. As an example, the first gene will represent the generator which is used for supplying the market first; the second gene will represent the second generator selected for supplying the remaining potion of the market. This process is repeated until the total system demand for that hour is met by moving from the left most gene to the right. The chromosomes for each subsequent hour are begun with the chromosome for the previous hour. This process will not only minimize the randomness introduced into the generator selection process but also will minimize the number of generators switched on and off from one market hour to the next. In most cases only a few allele values are modified to account for the demand variation from the previous hour to the hour being considered. An example to illustrate this scheme is presented below. Assuming a power system with 6 generators, the first step is to select the order in which generators will be selected to supply the demand. This order will be determined randomly. In this example, for market hour 1 the generator order will be chosen at random as 3, 4, 2,1 , 6 and 5 for one chromosome, while a second chromosome will be 3,2,4,1,5,6. The next step is to change the allele values of each of the genes to represent the output from the generators. These allele values will then be populated with appropriate generator-output values within their economic maximum and minimum limits. In this example, for chromosome 1, the corresponding outputs could be 100, 40, 60, 70, 0, 0 assuming a total system load of 270MW. For chromosome 2 one possible allele configuration could be 100, 60, 40, 70, 0, 0. Assuming that the system demand increased to 300 MW for market hour 2, one possible adjustment to chromosome 1 for hour 2 could be 100, 40, 60, 70, 30, and 0. Alternatively, chromosome 1 could be 100, 40, 60, 80, 20, 0 or 100, 40, 60, 70, 20, 10, these being other possible candidate solutions that are in the neighborhood of the original solution. The values selected for each allele will be based on the maximum and minimum economic limits of

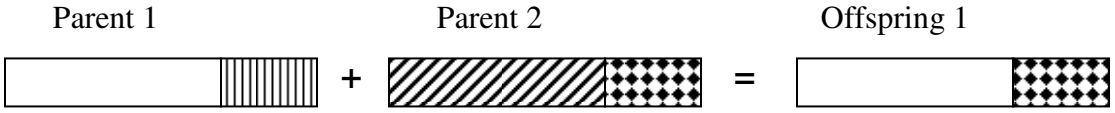
the generator to be used and also based on the step by which each generator could be moved up or down from its current operating point. Extending the examples above, since generator 3 is selected as the first gene of chromosome 1, the allele value to be populated will be selected at random, based on the fact that generator 3 has an economic minimum limit of 30 MW and an economic maximum limit of 150 MW; a value of 100 is selected at random, and the same process is repeated for generators 4, 2, 1, 6 and 5. When each subsequent generator is selected, the output assigned to it will be determined by its minimum and maximum operating limits along with assuring that the output selected does not exceed the total system demand to be supplied. Typically the output values selected from each generator are chosen to be multiples of 10 MW.

6.2.2 Crossover schema

For simplicity, a single crossover scheme was adopted in the solution formulation. Furthermore, the crossover scheme included a single crossover point as well. The location of the crossover point was selected randomly at the beginning of the study and this value was kept constant in all subsequent crossover operations when forming a new population from a parent generation. To illustrate the notion of the crossover point, a crossover point of 2 would indicate that the last 2 genes of a given chromosome will be interchanged with the last 2 genes of another chromosome, while a crossover point of 5 would indicate that the last 5 genes of a given chromosome will be interchanged with another compatible chromosome and its last 5 chromosomes. In order to ensure that the chromosomes in a given population have a sufficient number of partners to crossover, at first, the generators that are to the right of the crossover point will be selected based on their incremental costs. The generators which are most expensive will be used for making up this section. The next step would be to eliminate those generators that were chosen to form the right portion of the chromosome and select the remaining generators to form the left half of the chromosome. As an example, if we consider a power system with 10 generators and a crossover point of 4, the 4 most expensive generators will be chosen for the last 4 genes of the chromosome. The remaining 6 generators which are all different from the generators to the right of the crossover point are then reserved for the left half of the chromosome. This assignment will then assure a population that has chromosomes that can crossover with one another. Extending the example with 10 generators, if generators 2,3,5,7 are found to be the most expensive in the power system, they will be chosen for populating the right

halves of the chromosomes in a population. Then generators 1,4,6,8,9,10 will be automatically left for selection for the left halves of the chromosomes. Now any chromosome that is formed by adopting the approach described above would yield a chromosome that is compatible for crossover with another chromosome that is constructed in the same manner in the same population. This approach can be graphically shown in Figure 6.1.

Figure 6.1 - Crossover to form and Offspring



Here, an offspring can be formed from a pair of parents using the genes up to the crossover point from one parent and with the genes beyond the crossover point from the second parent. Typically, a high crossover rate is chosen to ensure that the solution traverses a sufficient portion of the solution space. Typically it is recommended that 85%-95% of the members from the current generation be used for crossover operations when creating a new population. The remaining 5%-15% of the original chromosomes are allowed to propagate to the next generation without any crossover. As an example, if a crossover rate of 85% is selected with population size of 200, only 170 chromosomes will be selected for crossover. The remaining 30 parent chromosomes are allowed to migrate to the next generation without any crossover. The crossover process will determine the generator selection order. As an example, if one of the parents has a generator selection order of 3, 1, 2, 4, 6, 5 and the second parent has the order 3, 2, 4, 1, 5, 6 with the crossover point selected to be after gene 4, the resulting offspring chromosomes would be 3, 1, 2, 4, 5, 6 and 3, 2, 4, 1, 6, 5, respectively. Once the crossover process is completed, output of each generator is checked to ensure that its limits are not violated. Using the above generator loading order, let us assume generator outputs are 30, 40, 50, 50, 30, 20 in the first chromosome and 30, 50, 40, 50, 40, 10 in the second chromosome. Even after the crossover, the generator loading pattern in each chromosome is maintained at the previous levels. With the revised generator order, if the output of generator 5 in chromosome 2 is not adjusted appropriately, the economic maximum limit of 30MW in this generator will be

exceeded. In order to ensure that the generators are operated within their economic limits, the output pattern in chromosome 2 will be adjusted. Thus, the first child chromosome with crossover would not require any adjustments to its generator outputs while the second chromosome would need an adjustment to make the chromosome become 30, 50, 40, 50, 30, 20. Reduction of 10 MW in the fifth generator is compensated by an increase of 10 MW in the sixth generator. Given that a 24 hour day-ahead market is considered, the same generator selection order is used for every hour of the day unless the selection is unable to supply the system demand. This approach would ensure that the least number of generators would be turned on and off throughout the day. Also with this approach the market would transition from the previous hours and day without major deviation in the selected generator operating schedule for the day. If the generator selection order as proposed by a given chromosome is unable to supply the system load in any one of the market hours, this chromosome will be discarded since it is no longer a viable solution for supplying the market.

6.2.3 Mutation Scheme

At the conclusion of the crossover, a selected sample of offspring chromosomes is chosen for mutation. In order to limit the level of randomness introduced into the solution process, the mutation scheme was kept constant from one generation to the next. Experts have recommended a mutation rate between 0.5%-1.0% as suitable for solving practical problems. A mutation rate of 0.8% was selected for the test cases analyzed in this research. With an appropriate mutation rate selected, the next step is to identify a mutation technique. An approach identified as *order-changing mutation* was adopted in this research. In this method the gene order between a pair of genes is reversed to form a new mutated offspring chromosome. Considering a day-ahead market with 9 generators available, and selecting only 0.8% of the total number of child chromosomes to be mutated, 2 child offsprings from a population of 250 will be subjected to mutation. The order-change scheme is kept constant from one population to the next. An example of the order-changing mutation is shown below. In this example gene 2 is swapped with gene 7 to form a new chromosome. With position swapping the order in which a given generator is committed to supply a given day-ahead market is determined. As described in Chapter 5, a random process is used to assign generator output values for the chromosomes selected for mutation. However,

given that the output levels of all other generators are not altered during mutation, the total output from these two generators before and after mutation will be maintained to ensure that all generators collectively are able to supply the system demand. Moreover, the allele values for each of these genes will have to be selected within the respective minimum and maximum economic limits of each generator occupying the given gene. In order to ensure that chromosomes resulting from mutation still would allow for crossover in forming subsequent populations, the pair of genes selected for mutation has been fixed to the left of the crossover point. The example presented below considers a mutation process for a chromosome where the crossover point was 2 or after the 7^{th} gene of the chromosome. During the mutation process the values that represent the generator output also have been adjusted as the generator order was reveresed.

(100 **20** 30 40 50 60 **80** 90 70) => (100 **60** 30 40 50 60 **40** 90 70)

6.2.4 Selection Scheme

A criterion for identifying the best chromosomes from the existing population to be retained in the next population is needed as part of the *GA* algorithm. The method adopted to retain the best chromosomes from one population to the next is identified as the *selection scheme*. The parent chromosomes that are retained from one generation to the next are considered as elite chromosomes. The percentage of elite chromosomes in conjunction with the crossover proportion and the mutation proportion governs how a new population is formed. Optimal selection of these parameters increases the performance of a *GA*.

In the combined algorithm, the original population is selected randomly. Subsequent populations are formed using an evolutionary process based on elitism, crossover and mutation principles. Next the fitness of the new population is compared with the original population. The abilities of a tabu search algorithm to retain the memory of good and taboo neighborhoods are utilized to ensure that revisiting bad neighborhoods is avoided. This is one of the basic differences between the combined algorithm and the *MOTS* algorithm.

6.3 *NSGA II* Solution Algorithm

As presented in the previous section, the *NSGA II* is a very specialized genetic solution algorithm. The following sections describe the parameter selection criteria for the *NSGA II* algorithm when used for solving the day-ahead market dispatch problem.

6.3.1 Encoding Scheme

Similar to the *MOTS/GA* solution algorithm, the *NSGA II* algorithm requires the careful selection of the chromosomes to represent the given problem. Here too, a value-encoding strategy was used. Offer-levels of individual generators were used to construct the genes of a chromosome. Using an example of 10 generators participating in a given day-ahead market, a chromosome length of 10 with each allele representing the output from each generator was chosen. The entries in the chromosome represent power output of each generator for a given market hour. The same approach as described in the *MOTS/GA* hybrid solution was adopted when forming the candidate chromosomes.

6.3.2 Population Size

This parameter is the number of chromosomes considered in a single evaluation. As recommended by others, a population size of 200 was used in this research.

6.3.3 Generation Size

This is the total number of different generations into which a given population is allowed to evolve. As recommended by previous works, the generation size was set at 200.

6.3.4 Crossover Scheme

The value chosen for this parameter is very similar to the values chosen in the *MOTS/GA* algorithm. Based on the recommendations made by the developers of the *NSGA II* algorithm, a binary crossover scheme that is described in the example below was used here. The value chosen for all evaluations was 0.7, which is in the range [0.6-1.0] recommended by experts who have used this algorithm extensively.

6.3.5 Mutation Scheme

The values used were similar to those used in the *MOTS/GA* algorithm. Here, a very low proportion of mutation was used. As recommended by the developers of the algorithm, a mutation probability equal to 1/(number of real variables considered in each test case) was used.

6.3.6 Distribution Indices

As required by the *NSGA II* algorithm, indices to control the simulated binary crossover distribution and real-variable polynomial mutation distribution of a given population need to be defined. An index of 10 from the recommended range [5-20] was used for crossover distribution, while an index of 30, from the recommended range [5-50] was selected for mutation distribution in all the studies. The values selected were based on the recommendations of researchers who have used this algorithm.

An offspring using the above parameters can be found as follows. If a pair of parents y_1 and y_2 have lower and upper limits of y_l and y_u respectively. With a crossover distribution index of η_c , the resulting offsprings c_1 and c_2 can be found using the following formulae [18,19].

$\beta = 1 + [2/(y_2 - y_1)]Min[((y_l - y_1), (y_u - y_2)]$

$\alpha = 2 - \beta^{-(\eta_c + 1)}$

$\delta_q = (\alpha u)^{1/(\eta_c + 1)}$, where u is a random number within the range [0,1)

$C_1 = 0.5\{(y_1 + y_2) - \delta_q[y_2 - y_1]\}$

$C_2 = 0.5\{(y_1 + y_2) + \delta_q[y_2 - y_1]\}$.

If one of the above offspring C is selected to be mutated to form a new offspring mc, the following formulae can be used along with mutation distribution index of η_m,

$\delta = Min\{(c - y_l), (y_u - c)\}/(y_u - y_l)$

$\delta_r = [2u + (1-2u)(1 - \delta)^{\eta_m}]^{1/(\eta_m + 1)}$, where u is a random number within the range [0,1)

$mc = c + \delta_r(y_u - y_l)$.

An example would help better understand how each of these parameters is used for formulating children from a given pair of parents. Assume two parent values of 10 and 30 each with each upper and lower limit of 0 and 50, respectively. Using the above formulae,

$\beta = 1 + [2/(30 - 10)]Min[((10 - 0), (50 - 30)]$

$\beta = 2$

$\alpha = 2 - 2^{-11}$, with $\eta_c = 10$.

$\delta_q = 0.8$ with u = 0.1

$C_1 = 0.5[(10+30)- 0.8(30-10)] = 12$

$C_2 = 0.5[(10+30)+ 0.8(30-10)] = 28$

Now if C_1 is selected for mutation,

$\delta = Min\{(12-10), (30-12)\}/(30-10) = 2/20 = 0.1$

$\delta_r = [2x0.2+(1-2x0.2)(1-0.1)^{30}]^{1/31}$ with u = 0.2.

$\delta_r = 0.972$

$mc = 12 + 0.972x(50-0) \approx 49.$

6.4 Objective Functions

The key operational objective of the market-clearing problem is to operate the market at the lowest cost while ensuring that the companies which own generators do not resort to market manipulation activities. In order to realize these objectives, two competing functions will be simultaneously optimized. Given that the day-ahead market is constrained, conditions such as the minimum and maximum limits of the generators, ramp-up and ramp-down requirements of the generators and transmission-line limitations will also be considered in conjunction with the two functions that are optimized simultaneously. The inputs used for analysis are the system data including information on connections between different system buses with impedence values between them, selling offers from generators, buying offers from loads, and hourly load variations over the 24 hours for all the loads. The solution algorithm consider these inputs with the constraints defined previously to evaluate all possible combinations of outputs from generators to meet the load requirements for every hour of the day. From all the possible solutions those that are on the Pareto-front are suggested as the optimal solutions for the problem. Each solution consists of an hourly generation schedule for 24 hours of each generator to meet the load for each hour of the day.

6.5 Selection of Optimality from Multi-Objective Optimization Problems

Evolutionary multi-objective optimization is aimed at converging toward a non-dominant pareto-front. The pareto-front results in a set of solutions that are feasible, rather than a single final optimal solution, as would be found by single-objective optimization. However, given that all practical problems aim at finding a single optimal point of operation, an unnecessary burden is placed on the decision-maker, namely that of selecting from among the feasible set. The challenge of such a selection becomes even more difficult when the number of objectives increases and a large set of pareto-solutions is available for selecting one optimal solution. As done in the previous sections of this dissertation, an optimal operational point from each of the pareto-fronts was selected based on the industry recommendations. The decision criteria in those studies were to limit the overall average system-wide *DHHI* for the market-day to a threshold value of 1800. Although this is one approach for selecting the desired optimal operation point, a systematic method that validates the above selection is worth investigation. An advance in the field of many-criteria decision-making that has gained wide acclaim recently is proposed and used for this purpose. Work presented in [29], [30], and [31] has incorporated fuzzy-set theory and fuzzy dominance using linguistic knowledge of preferences by the decision-maker. Other work presented in [38] and [39] shows fuzzy membership functions as a tool for the numerical formulation and treatment of the dominance definition. Here, techniques for transforming qualitative relationships between objectives into quantitative attributes are presented. Also, influences of the decision maker's preferences are removed when selecting optimal operational points from multi-objective optimization problems.

A multi-criteria decision-making approach based on fuzzy-set theory as proposed in [31] was selected for aiding the decision-maker who is responsible for selecting the optimal operational point for the day-ahead market. The underlying mathematical constructs of this approach used in this work are presented below.

When two solutions v_1 and v_2 are compared with one another, based on pareto-optimality definitions, v_1 is considered dominant in a pareto-sense, if in all but one objective, it is better than v_2. However, when dealing with problems with many objective functions, a more general

definition of selecting a solution that involves comparing three possible outcomes to make the final determination will have to be formulated.

For each pair of points v_1 and $v_2 \in \Omega$, the function n_b counts the number of objectives where v_1 is better than v_2 since the resulting objective function value is lower for v_1 when compared with that for v_2. n_e counts the number of objectives where the solutions are equal to one another and n_w counts the number of objectives where v_1 is worse than v_2. The following formulae can be used to define the notions discussed:

$$n_b(v_1, v_2) := \left|\{i \in N \mid i \le M \;\; \& f_i(v_1) < f_i(v_2)\}\right|$$

$$n_e(v_1, v_2) := \left|\{i \in N \mid i \le M \;\& f_i(v_1) = f_i(v_2)\}\right|$$

$$n_w(v_1, v_2) := \left|\{i \in N \mid i \le M \;\; \& f_i(v_1) > f_i(v_2)\}\right|.$$

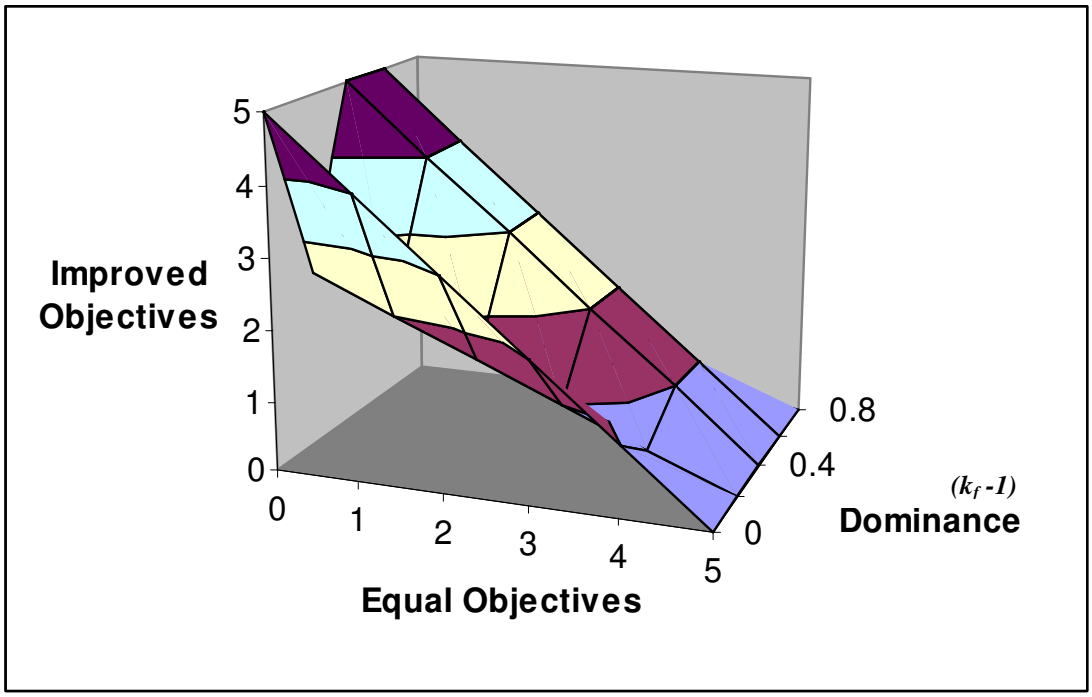
Where M is the number of objectives being optimized and f_i represents an objective function. Based on fuzzy arithmetic for a selected domain with M objectives, v_1 is said to (1-k_f) dominate v_2 if and only if:

$$n_e \le M$$

$$n_b \le \frac{M - n_e}{k_f + 1},$$

where $0 \le k_f \le 1$. With the above definition once k_f is provided it is easy to check whether a given candidate solution (1-k_f) dominates another solution. A plot of the number of improved objectives against the equal objectives for a 5 objective case and (1-k_f) dominance is presented in Figure 6.2 below. Using these numbers a decision-maker can select a suitable operating point among the multiple pareto-front solutions. Based on fuzzy math, for the same problem with 5 objective functions, a 0.25 dominance value means a candidate-solution from the pareto-set that is found to be better than another candidate-solution for 4 out of the 5 objectives could be chosen as the optimal operating point.

Figure 6.2 - Number of Improved Objectives Plotted Against the Number of Equal Objectives and (1-k_f) Dominance



CHAPTER 7 - Case Studies

In order to evaluate the suitability of the each of the solution algorithms for solving the day-ahead market problem, three test systems were investigated. The first test system comprised 5 generators and 3 loads. The second system comprised 10 generators and 6 loads. The third test system had 50 generators and 30 loads. Impacts of operating a power system without constraints on line-loading as well as with constraints on line-loading were simulated. Influences of market-power exhibition by individual market players on the overall day-ahead market dispatch problem were evaluated on the first two test systems using each of the chosen algorithms. A larger system comprising 50 generators and 20 loads was then used to evaluate the ability of each of the solution algorithms to scale up successfully and thereby demonstrate the practicality of using each of the solution algorithms for large power systems found in real life. The impact of market domination due to increasing the ownership share of one company was tested to validate the robustness of the *NSGA II* algorithm. Results from the three multi-objective-evolutionary algorithms (*MOEA)* algorithms were compared with one another and with the results from a traditional linear programming (*LP)* algorithm that is widely used in the industry at a specified industry-recommended market-power index. Benefits of using a fuzzy decision criterion to select a more suitable optimal point from the pareto-set rather than using a threshold value are also presented. The results and details of the studies conducted are presented in the ensuing sections.

7.1 Analysis of a 5-Generator, 3-Load Power System with No Market Power

The suitability of each of the solution algorithms for solving the day-ahead market dispatch problem was first evaluated using a power system consisting of 5 generators and 3 loads. The test system used for this analysis is depicted in Figure 7.1. In this case, all generators are assumed to be available for participation in the market, with no generators or lines taken off-line for maintenance. In the evaluations, the generators were assumed to have their own offer-curves with their own operational characteristics. The demands from each one of the loads at are also considered to be varying from one hour to the next. Since one requirement of any day-ahead

market is to dispatch generators at the lowest possible cost while fully recognizing the power system conditions, the market-dispatch problem under this scenario becomes a constrained economic dispatch.

The corresponding load profiles from each of the three loads used in the simulation for the chosen 24-hour period are shown in Figure 7.2. The offer-curves for the five generators considered in this study are assumed to be monotonically increasing as shown in Fig. 3 with economic minimum and maximum limits available for supplying into the market. Information on buying offers from the loads used in simulations are shown in Appendix B.

Figure 7.1 - 5-Generator, 3-Load, 10-bus Test System

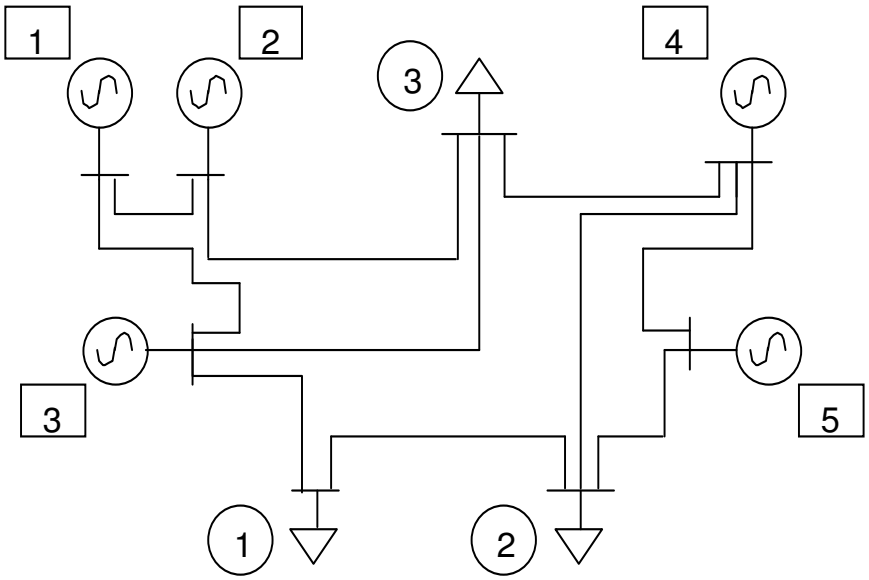


Figure 7.2 - Load Profile for the Market Day

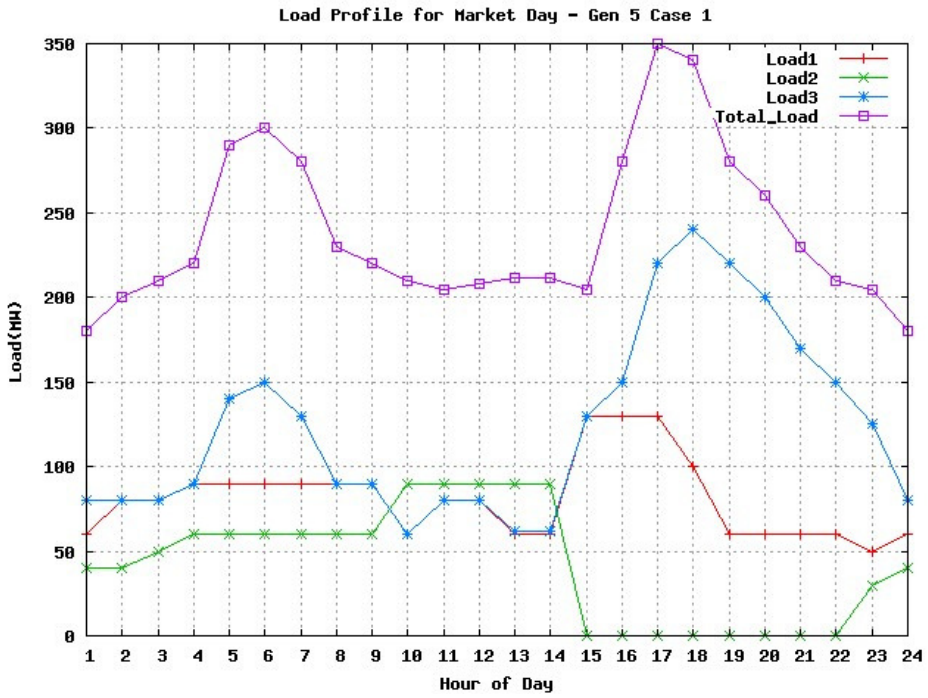


Figure 7.3 - Generator offer-curves

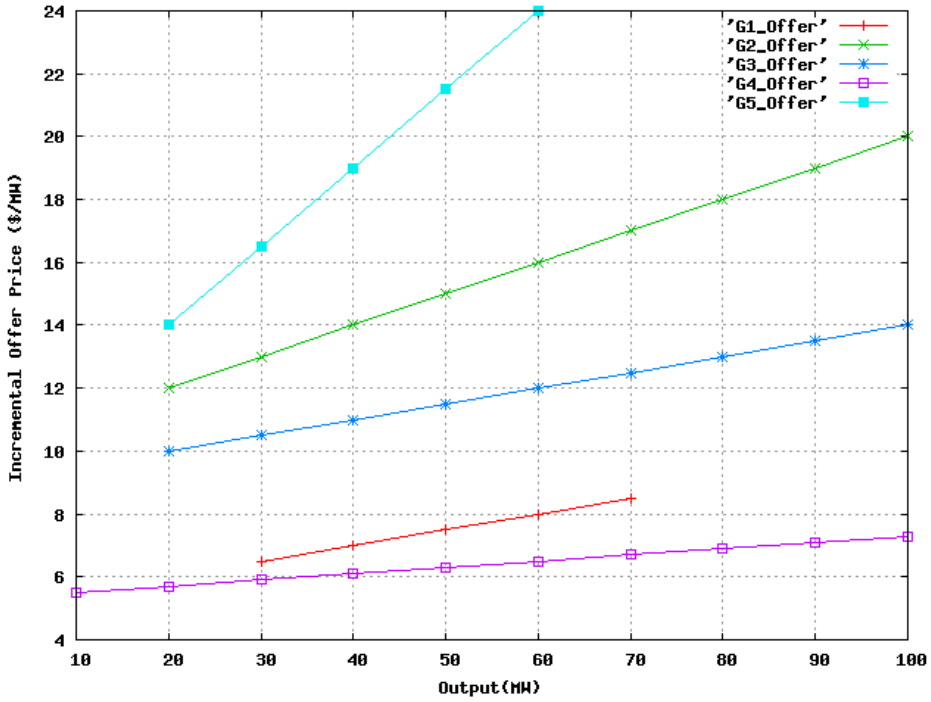
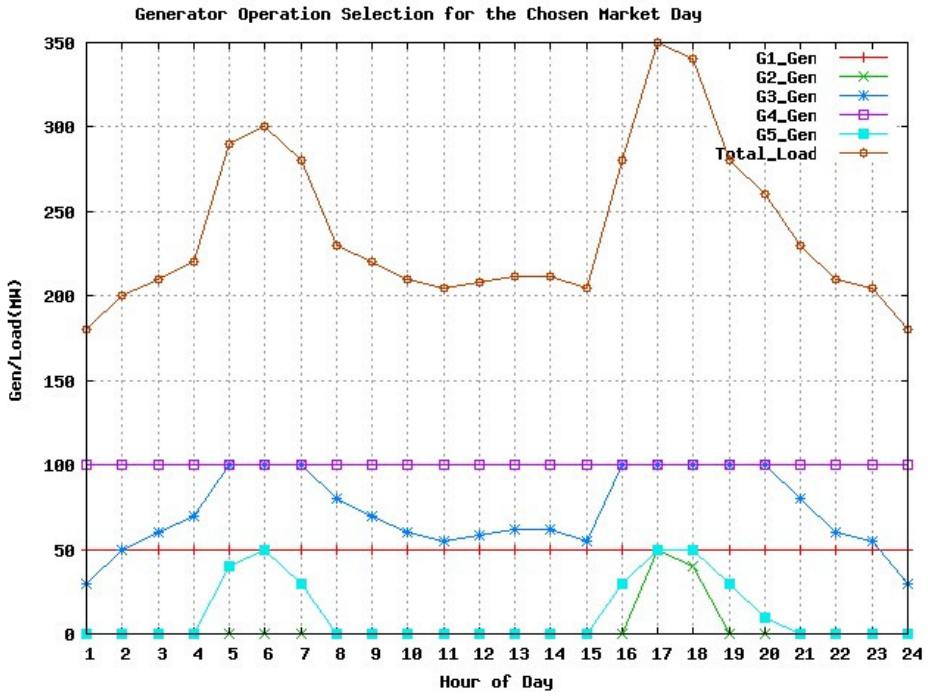


Figure 7.4 - Generator Operating Schedule



A power system with all generators available to offer to the market, with no system congestion, generator-limit violations or transmission-line-constraint violations present was simulated. The effects of market-power manipulation attempts by individual generator owners were disregarded in this study. Based on the above, the problem became a single objective minimization. Each of the solution algorithms was applied for solving this test case in order to evaluate its suitability. The multi-objective tabu search algorithm (*MOTS*) was used first. The generator-dispatch scheme to serve the total load for the market day as recommended by the *MOTS* algorithm is presented in Figure 7.4. The results show that a minimum number of generators were turned on and off throughout the market day.

From the results it is evident that generator 4, which is the most economical to operate, is chosen first, followed by generator 1. Both these generators are recommended to be operated throughout the day. Generator 3 which is the next most economical unit supplements generators 1 and 4 to meet the load profile changes. The other more expensive generators are selected to operate in specific multiple time periods. The same problem scenario was analyzed using the

two other algorithms: combined *MOTS/GA* algorithm and the *NSGA II* algorithm. The generation-dispatch levels recommended by each of the algorithms are seen to be identical. However, the time taken by each algorithm to arrive at the final solution was different. The total cost of operating the market for the selected market day was found to be $12,375.00. The *NSGA II* algorithm found the optimal solutions in the shortest time, followed by the *MOTS/GA* algorithm. The *MOTS* algorithm took the longest time to converge. These simulations were carried out on a 1.8GHz *Pentium IV* server running the *Linux* operating system. Simulations were carried out utilizing software that was developed using the ANSI C language. Since results from each of the algorithms were identical, one can conclude that any one of these algorithms can be used successfully to solve the simple 5-generator day-ahead market. A table comparing results from using each of the three algorithms for this test case is presented below.

Table 7.1 - Performance comparison between algorithms

Algorithm	# of Trials/ Generations to find solutions	CPU Time for solution (Sec.)
MOTS	657	2.15
MOTS/GA	421	1.76
NSGA II	497	0.95

7.2 Analysis of a 5-Generator, 3-Load power System with Congestion and No Market Power

Next, the same power system was used to evaluate a day-ahead market when a transmission line has reached its thermal limits. The investigation here was to evaluate the suitability of each of the algorithms when an additional constraint is introduced into the solution space. In order to create a transmission-line over-load constraint condition, the load profile for the entire market day was increased by 15% from the previous case. This forced the transmission line between generator 4 and load 3 to reach its thermal limit. As in the previous analysis, each of the algorithms was capable of finding an optimal operating solution for this scenario. The resulting operational scheme for every hour of the selected market day using the *MOTS/GA*

algorithm is depicted in Figure 7.5. Looking closer at the results, it is notable that the minimum operating limit of generator 5 is maintained during the hours of 7 and 16.This demonstrates that the dispatch scheme found by the algorithm has ensured that the minimum-operating-limit constraint of the generators is enforced. It is also evident that although generator 4 is one of the more economical units, it was unable to offer its full capacity into the market due to the line constraint existing in the system. This shows that the algorithm has successfully enforced the influences of branch-capacity limit constraints discussed in chapter 3. The corresponding results using the *MOTS* and *NSGA II* algorithms for the same test system are presented in Figures 7.6 and 7.7, respectively. From the results it is evident that all three algorithms dispatched generators 1 and 4 at the same level. Also notable is the fact that all algorithms enforced the branch-capacity limit constraint by partially dispatching generator 4. The dispatch schemes recommended for the next 3 most expensive units varies from one algorithm to the other. Also notable from the results is the fact that each algorithm ensured that the minimum operating limits of the generators are maintained when finding the generator dispatch schedule as described above for the *MOTS/GA* results. The results confirm that each of the algorithms is successful in finding an optimal constrained economic dispatch scheme for the day-ahead market. From the results presented in Table 7.2, the optimal costs found by each of the algorithms are seen to have marginal cost differences explainable by the differences in the dispatch schemes between the algorithms. The lowest operational cost of $12,382.00 was obtained by the *MOTS/GA* algorithm. The next lower cost of $13,435.00 was obtained by the *NSGA II* algorithm with a difference of $1053.00. The highest operational cost was obtained by the *MOTS* algorithm. This cost was $1140.00 higher than that obtained from the *NSGA II* algorithm. Also, as demonstrated in the table, the *NSGA II* algorithm took the shortest time to find its optimal solution.

Table 7.2 - Comparison of results between algorithms

Algorithm	# Of Trials/ Generations to convergence	CPU Time (Sec.)	Total Minimum Operational Cost ($)
MOTS	921	2.96	14,575.00
MOTS/GA	756	1.96	12,382.00
NSGA II	524	1.60	13,435.00

Figure 7.5 - Generation operation selection in the presence of congestion using *MOTS/GA* algorithm

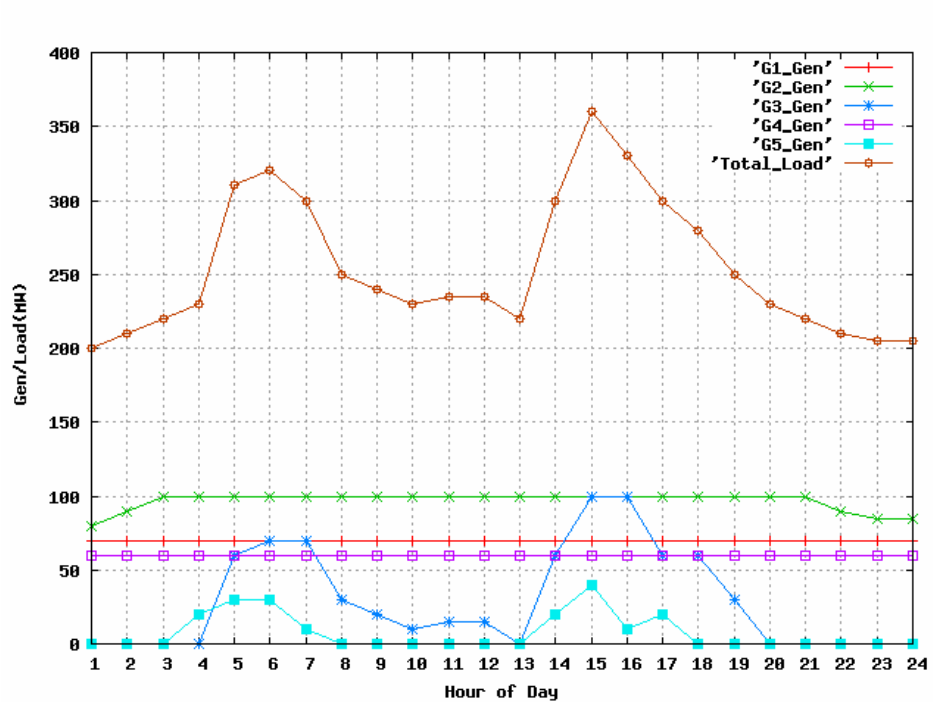


Figure 7.6 - Generation operation selection in the presence of congestion using *MOTS* algorithm

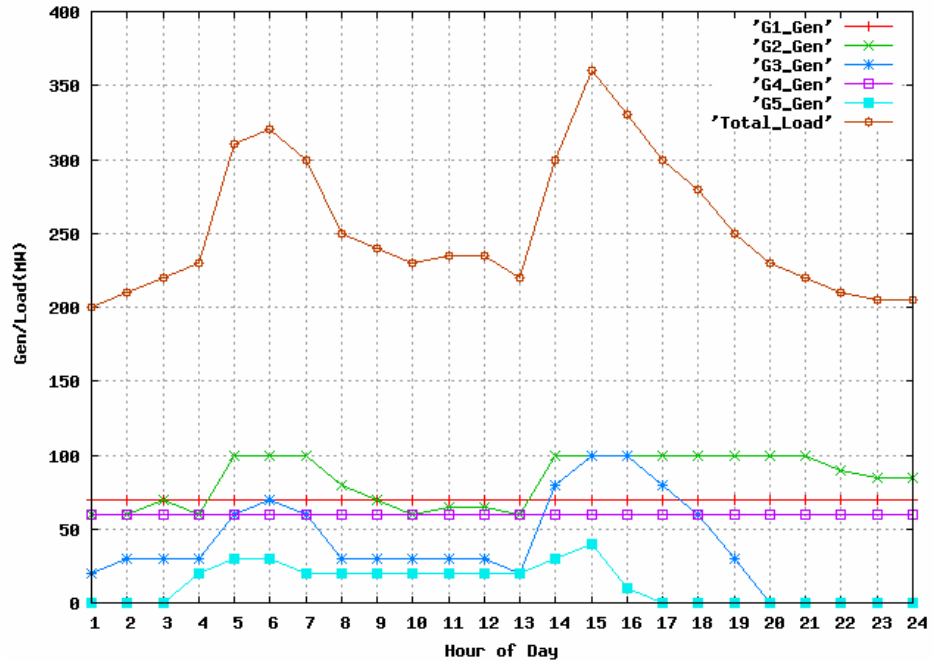
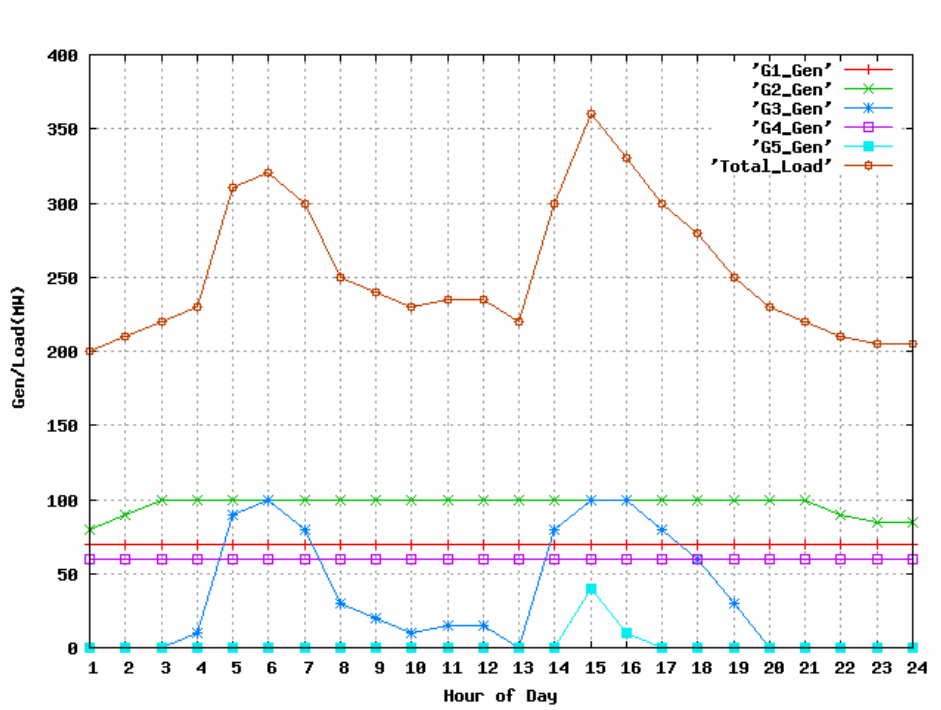


Figure 7.7 - Generation operation selection in the presence of congestion using *NSGA II* algorithm



7.3 Analysis of a 5-Generator, 3-Load power System with Congestion and Uncontrolled Market Power of owners

The next step was to extend the scope of the study to incorporate the notions of market power exhibited by companies. Here, the test was to evaluate whether companies owning generators can exercise market power to their advantage. The solution space now becomes that for optimizing two competing objective functions: how to minimize the overall market operational costs, while minimizing the opportunity for market-power exhibition by individual generator-owning companies.

In order to simulate such a scenario, generators 3 and 5 were assumed to belong to the same company. The incremental costs of generators 3 and 5 were left unaltered from the previous case, with generator 5 being the most expensive unit offered to the market. The load profile was kept as it was in the previous test case, hence the congestion on the transmission line between generator 4 and load 3 was included in the analysis. In the solution process, the *Dynamic Herfindhal-Hirschman Index* (*DHHI*) for the entire market day was calculated, concurrently while selecting the optimally economic dispatch scheme.

Since the problem involves optimizing two competing objective functions, a set of non-dominant solutions is obtained. The pareto-optimal front graphs obtained from each of the solution algorithms are shown in Figure 7.8. Comparing the three graphs obtained from each of the algorithms, one can clearly see that the *NSGA II* provides a solution set superior to those of the other two because its pareto-front lies below the other two fronts. It is also notable that this algorithm took the least time to arrive at the final pareto-front, as shown in Table 7.3. As one can see from the graphs, the pareto-fronts from the other two algorithms are in the same general neighborhood. Since the differences in the three algorithms are not significant, any one of the three algorithms can be used for solving the given market problem. Since the solutions now take the form of a solution vector rather than a single optimal value (as would be obtained in a single optimization problem), the challenge is to select the best operation point for clearing the day-ahead market. Since there was no attempt to minimize the market concentration in this

evaluation, one could choose the lowest market operational cost point for clearing the market. However, the *ADHHI* calculated for the entire system by considering the corresponding index for every hour of the day at the lowest optimal cost from the pareto-front for the *NSGA II* algorithm is seen to be significantly above the acceptable market concentration index of 1800 recommended by the Department of Justice. This calls for further scrutiny of the results obtained for the systems for the entire market day.

The corresponding plots of the *DHHI* changes for the market day for the entire system, for generators 3 and 5 combined and the *DHHI* curves for the 3 other independent companies is depicted in Figure 7.9. From this graph one can see that for all hours of the day, the system-wide *DHHI* exceeds the U.S. Department of Justice recommended market-concentration-value of 1800. The average *DHHI* for the 24-hour market day based purely on economic dispatch and unit commitment is seen to be 3569. Since this value is significantly higher than the perceived moderate market concentration threshold, one can conclude that the generator owners will have opportunities to exercise market power. Investigating further, the individual average *DHHI* for the company owning generators 3 and 5 is found to be 858. This number is clearly below the market concentration threshold. The resulting market concentration values for generator 2 and generator 4 show the owners could use their market share to influence the market. Since the overall *DHHI* is above the recommended market concentration threshold, pro-active monitoring to watch for sudden changes in the offer-prices of the generators owned by the same company is recommended to ensure that participants do not take undue advantage of the market condition. Table 7.3 provides a good comparison between the results based on the final dispatch-schemes that were chosen from each of the pareto-front graphs obtained from the 3 algorithms.

Table 7.3 - Comparison of results between algorithms

Algorithm	#Trials/ Generations to converge	CPU Time (Sec.)	Total Minimum Cost ($)	Average *DHHI* /Hr at Min. Cost
MOTS	2096	3.50	14,996.00	4865
MOTS/GA	1867	2.40	14,725.00	4432
NSGA II	2034	2.10	14,526.00	3569

Figure 7.8 - Pareto optimal front graphs under the different solution algorithms

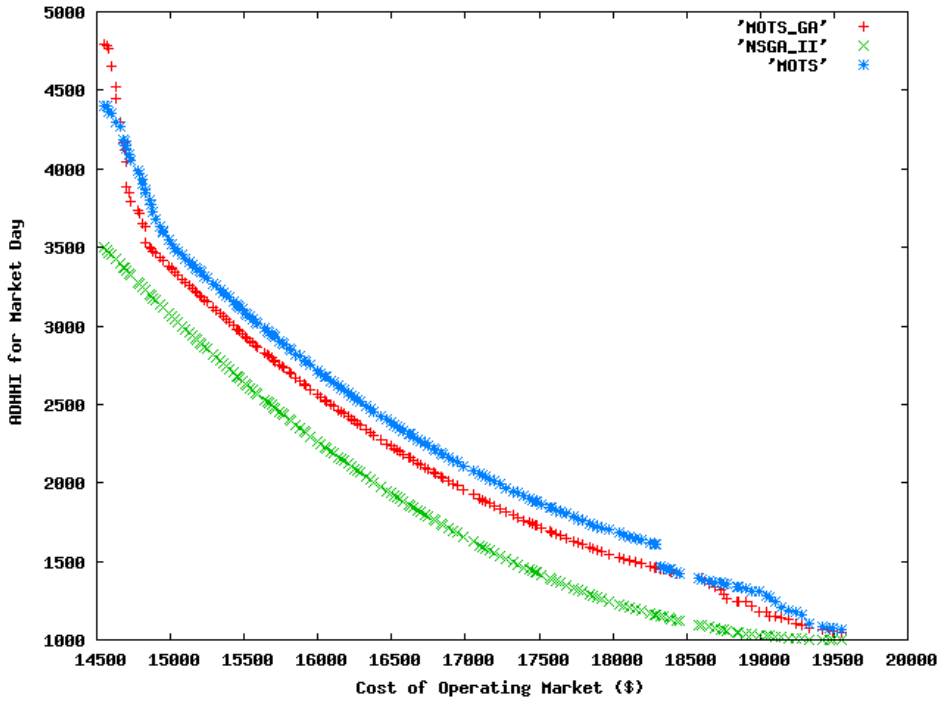


Figure 7.9 - *DHHI* chart for 5-generator test case using *NSGA II* algorithm

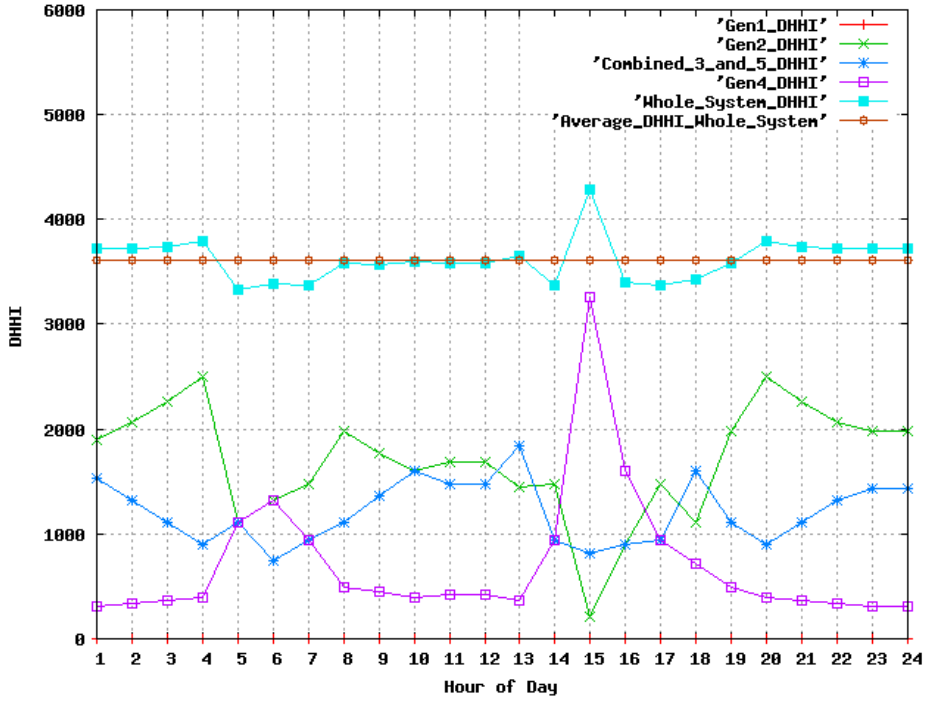
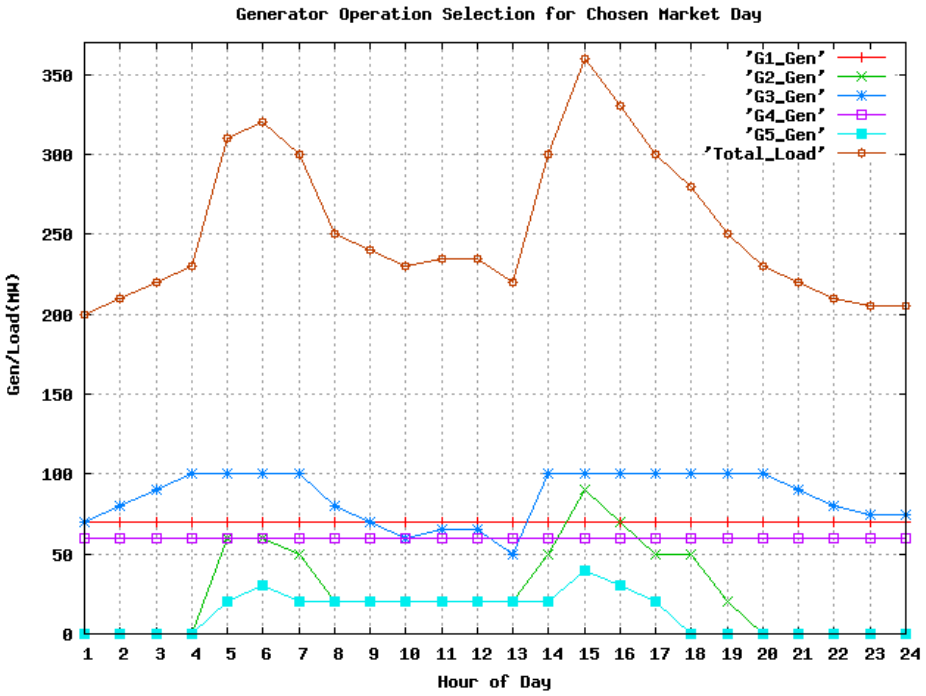


Figure 7.10 - Generator Operating Schedule as found by *NSGA II* algorithm



7.4 Analysis of a 5-Generator, 3-Load Power System with Congestion and Mitigated Market Power of Owners

How the company owning generators 3 and 5 uses its strategic position to its advantage to manipulate the market was considered next. In this scenario, it is assumed that the offer-prices of units 3 and 5 are increased from their previous values, with the power system conditions and the load to be served remaining unchanged. The same transmission-line-limit constraint prevails for this study as well. The generator offer-curves used for this scenario are shown in Figure 7.11.

As done previously, all three algorithms were used to solve for optimal cost of operating the market while ensuring that the market power is controlled. The resulting pareto-optimal front graphs from each of the algorithms are presented side by side in Figure 7.12. These graphs showthat the pareto-front obtained from the *NSGA II* algorithm lies closest to the origin. This indicates that this algorithm offers the best set of solutions to be used for operating the market. With the goal of limiting market power exercised by generator-owning companies, and selecting a threshold value of 1800 for the *DHHI*, the non-dominant solution point with an average *DHHI* of 1802 at an overall cost of $16,487.00 was chosen from the optimal set derived from the *NSGA II* algorithm. Figure 7.13 shows the generator-dispatch scheme obtained from this operating point. The fact that the two most economical generators, namely generators 1 and 4, were chosen first to supply the system load is evident from the results. However, generator 4 is able to offer its capacity partially due to the line-constraint condition, which is preventing it from dispatching up to its economic maximum limit. One can see that the outputs from generators 3 and 5 have been reduced considerably from the previous test scenario to ensure that the market manipulation capabilities of these two generators are contained. In order to ensure that the total system load is met, generator 2 is selected to operate throughout the whole day.

Figure 7.11 - Generator Offer Curves

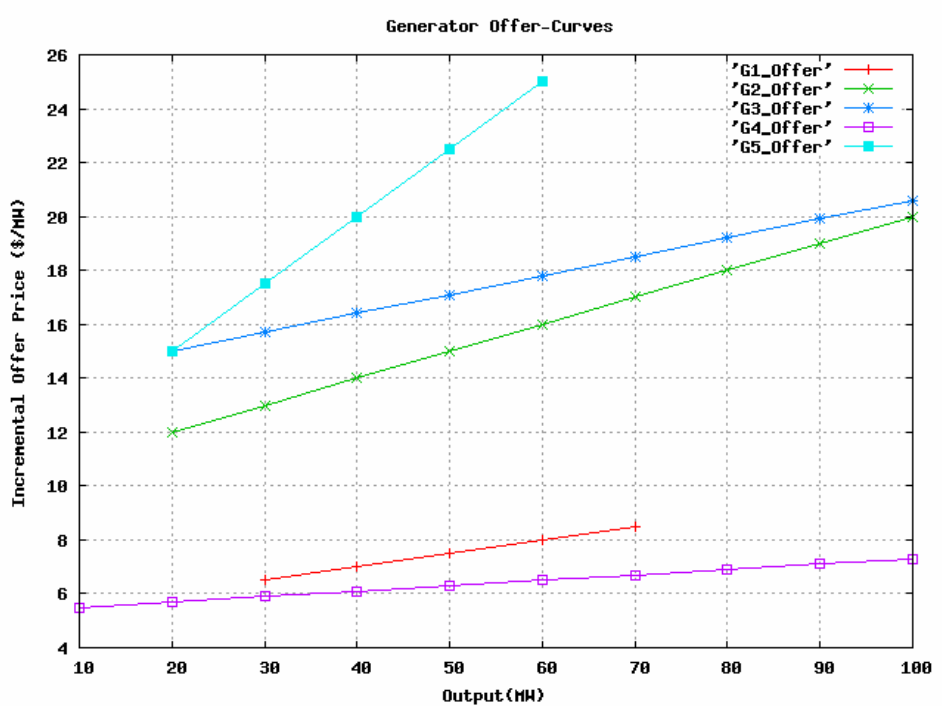


Figure 7.12 - Pareto Optimal Front Graphs

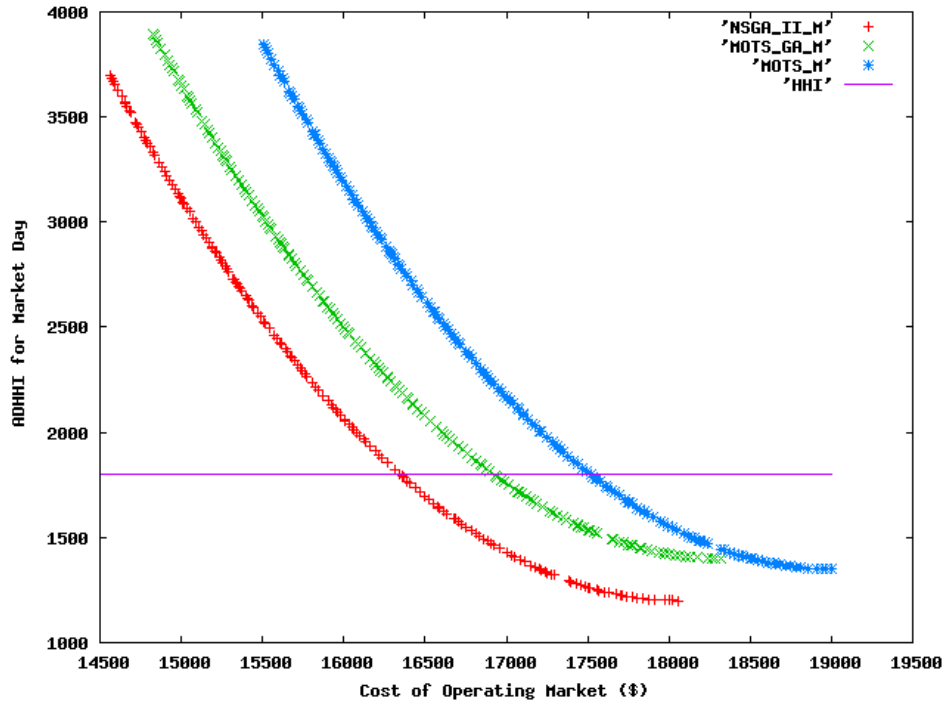


Figure 7.13 - Generator operation match-up

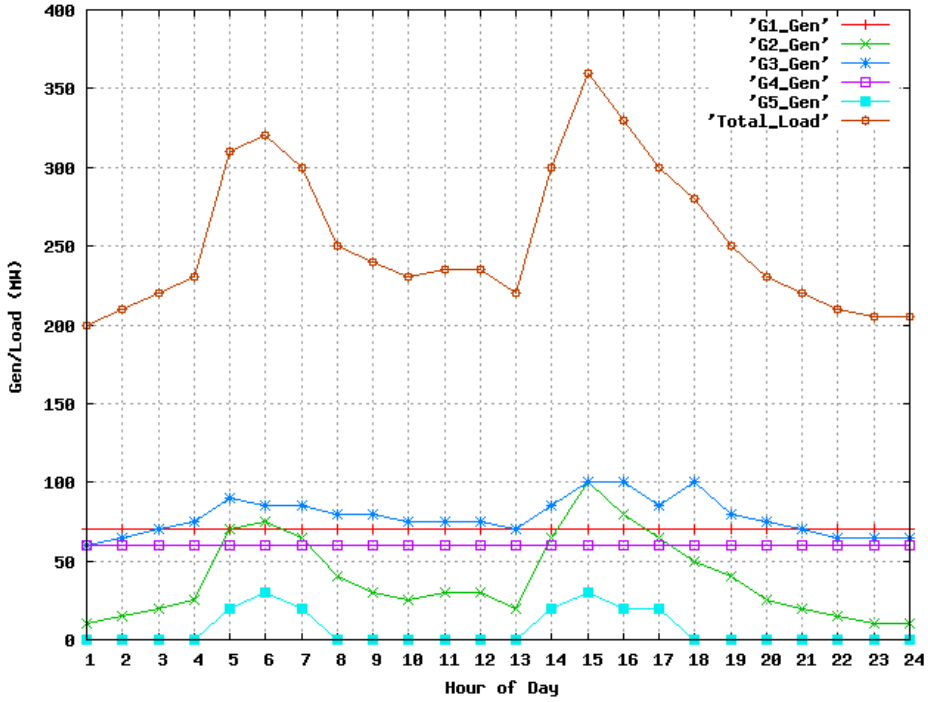
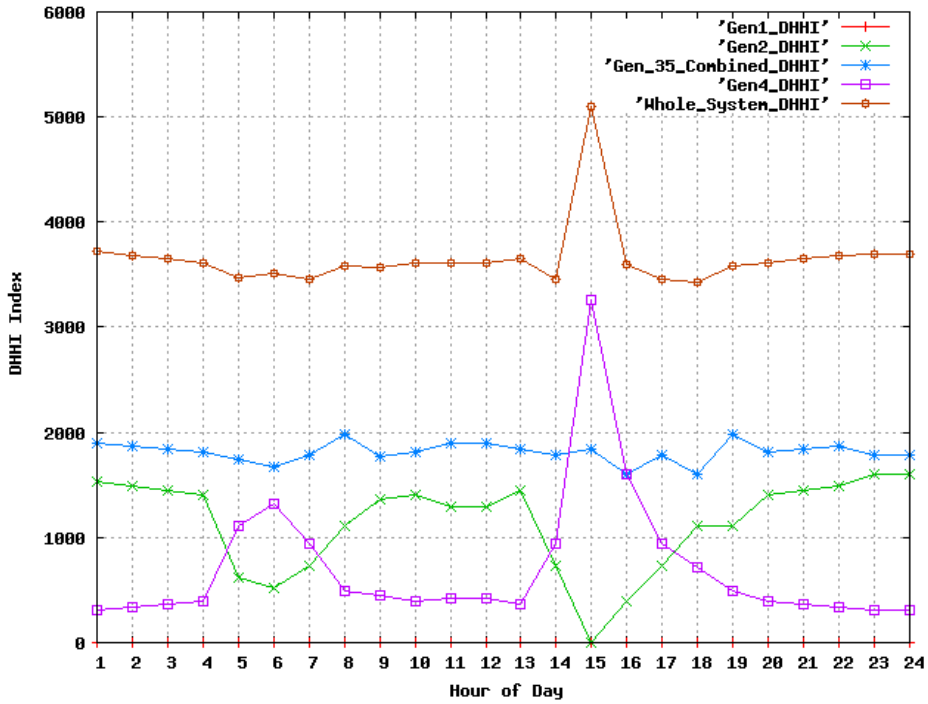


Figure 7.14 - *DHHI* chart for 5 generator test case using *NSGA II* algorithm



As Figure 7.13 shows, in order to inhibit the company owning generators 3 and 5 from using its market concentration to its advantage, it has been forced to turn off unit 5 between the hours of 1-4, 8-13 and 18-24, respectively, so as to minimize the average *DHHI* for the market day. The changes in *DHHI* throughout the day for the entire power system, generators 3 and 5 combined, together and that for each independent generator is depicted in Figure 7.14. The results show that the *DHHI* of all companies are below the recommended threshold of 1800 with the exception of hours 8 and 19 for the company owning generators 3 and 5. The line-constraint condition limiting generator 4 from dispatching beyond 60 MW has forced upon it a *DHHI* higher than the 1800 threshold value.

Key performance charachteristics for each of the algorithms are provided as a comparison in Table 7.4. In order to compare the process presented in this dissertation with the currently used practice of solving the day-ahead market minimizing only the cost, the problem was solved using a Lagrangian-relaxation-based Linear Programming (*LP*) method [11]. The same constrained power system under the same operational conditions was used for this analysis, so that the results can be compared effectively. The results from the *LP*-based method are shown in row 4 of Table 7.4 below. The corresponding *DHHI* at the optimal economic dispatch was also computed for comparison purposes. The minimum operational cost for operating the market as recommended by the *LP* method is clearly below the values chosen by the other three algorithms. However, the resulting average *DHHI* of 3870 for this case shows that the owner of generators 3 and 5 has used its market concentration to its advantage. It is also noteworthy that the *LP* algorithm took over three times as long as the other methods to find its final solution.

Table 7.4 - Comparison of results between algorithms

Algorithm	# Trials/ Generations to Converge	CPU Time (Sec.)	Selected Total Minimum Operational Cost ($)	Selected *DHHI*/Hour
MOTS	2645	4.70	17,484.00	1802
MOTS/GA	1984	4.20	16,889.00	1802
NSGA II	2156	3.90	16,487.00	1802
LP	7645	12.40	14,974.00	3870

In order to verify the repeatability, stability and robustness of a solution algorithm, the effect of the initial solution on the final solution was evaluated. A different initial solution is obtained by using a different random seed in each solution. For this, the same test scenario was analyzed 10 times using different starting solutions, while keeping all other operational constraints and system conditions constant. The *NSGA II* algorithm, which yielded the best minimum operation cost for the test case, was used for this evaluation. The minimum costs predicted for each run by *NSGA II* differed from one another by $47.00 or less, with a largest minimum operating cost of $16,496.00 and a smallest minimum operating cost of $16,449.00. Under all these simulations the *DHHI* was held around 1800. The results of each of these independent runs are presented in Table 7.5 below.

Table 7.5 - Results from repeated simulations

Random Seed Used	# Trials/ Generations to Converge	CPU Time (Sec.)	Selected Total Minimum Operational Cost ($)	*DHHI*/Hour
0.1	2157	3.80	16,452.00	1803
0.2	2159	3.90	16,478.00	1804
0.3	2156	3.70	16,449.00	1802
0.4	2153	3.60	16,456.00	1803
0.5	2156	3.50	16,478.00	1800
0.6	2159	3.70	16,496.00	1801
0.7	2156	3.90	16,487.00	1802
0.8	2157	3.60	16,463.00	1801
0.9	2155	3.70	16,459.00	1804
0.95	2158	3.90	16,479.00	1804

7.5 Analysis of a 10-Generator, 6-Load Power System with Congestion and Mitigated Market Power of Owners

In order to ensure that the solutions obtained previously by applying each of the three algorithms have no dependency on the size and nature of the power system, a second sample power system comprising 10 generators and 6 loads was chosen. A schematic of this power system is shown in Figure 7.15. The corresponding offer-curves for the 10 generators are shown in Figure 7.16. Profiles of the 6 loads in the system are shown in Figure 7.17 and information on their buying offers is included in Appendix B . In order to learn the impacts of market power when a given company owns a majority share of the market, generators 3 and 5 were assumed to be owned by a single company, while the remaining 8 generators were assumed to be owned by different independent companies with each company owning one generator.

Figure 7.15 - 10-Generator, 6-Load, 10-bus test system

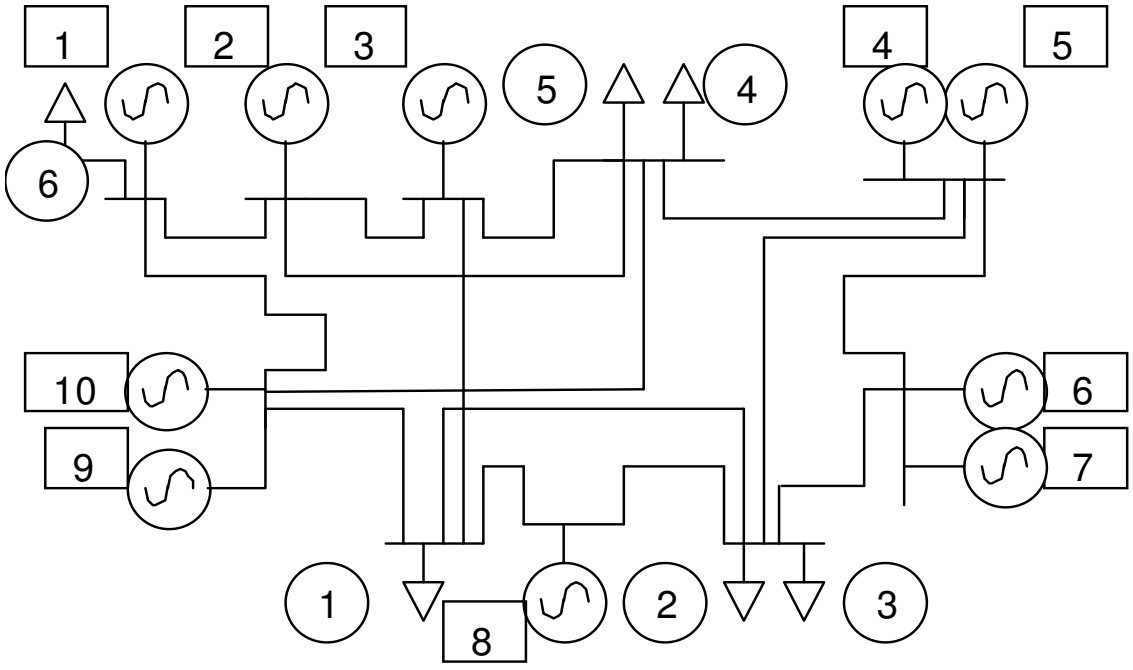


Figure 7.16 - Generator offer-curves for the 10 generators

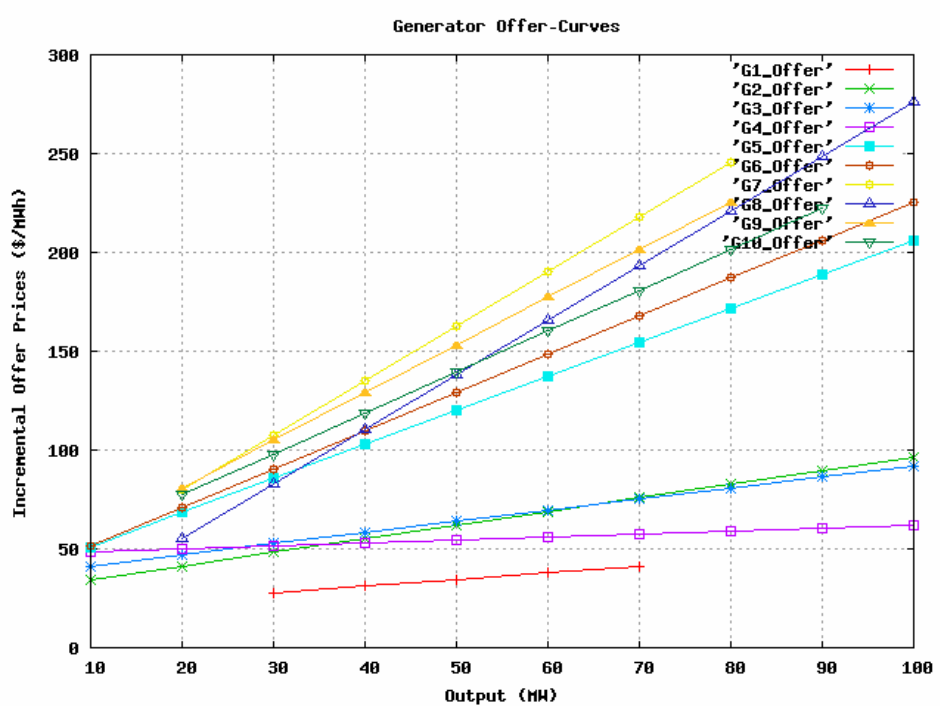
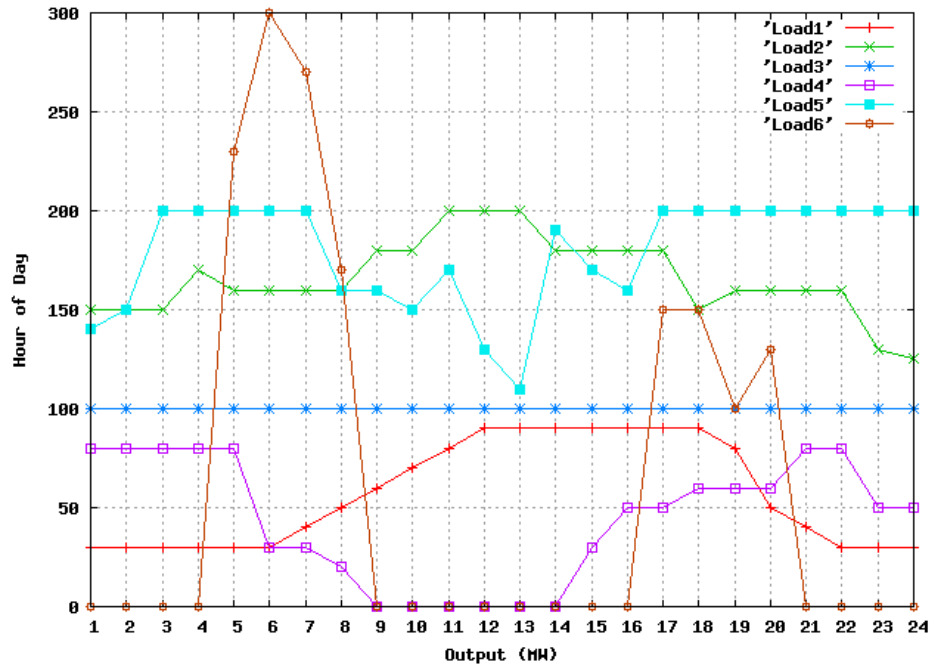


Figure 7.17 - Load Profiles



All three algorithms were applied to find the lowest operating cost of a given market day while minimizing the market power exhibited by individual generator owning companies. The final pareto-front graphs found from each of the three algorithms namely *MOTS, MOTS/GA* and *NSGA II* are presented in Figure 7.18. The progression of pareto-fronts towards their final front at 500, 1000, 1500 iterations when the *NSGA II* algorithm was used is presented in Figure 7.19. The fact that the pareto-fronts moved from right towards the origin as the solutions progressed demonstrates the improvements the algorithm achieves with each iteration, moving toward the pareto-solution-set that is not dominating either of the two objective-functions being optimized. A plot of minimum values of each objective function at different iteration-count values is presented in Figures 7.20 and 7.21, respectively. These graphs show the improvement in each objective function as the solution process progressed. The relative improvement achieved after the 1220^{th} iteration for each objective function is seen to be very small. As a comparison, the values in Figures 7.20 and 7.21 correspond to the minimum values of the pareto-front at a given iteration cycle shown in Figure 7.19, while the final pareto-front graph presented in Figure 7.18 is the same as that is presented in Figure 7.19 for the *NSGA II* algorithm. As an example, the minimum optimal cost of operating the market after iteration 1500 is $20430.00, while the corresponding *DHHI* value after this iteration is 1602. These two values are the corresponding lowest optimal operating cost and the lowest *DHHI* value from the pareto-front found after 1500 iterations in Figure 7.19.

Figure 7.18 – Pareto-optimal front graphs for different algorithms for the 10-generator test case

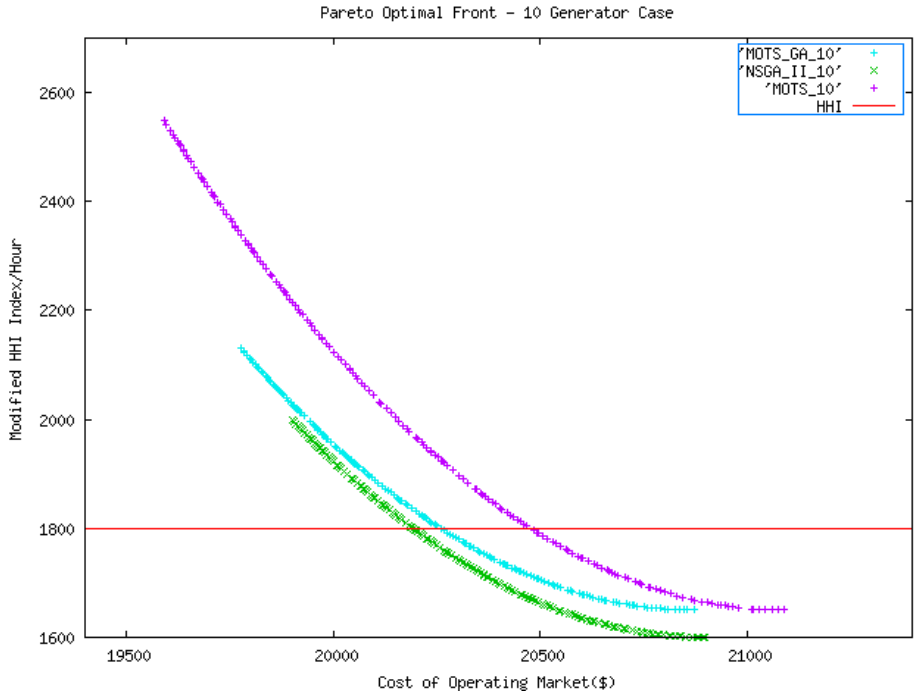


Figure 7.19 – Pareto-optimal fronts at different iterations using *NSGA II* algorithm, 10-generator test case

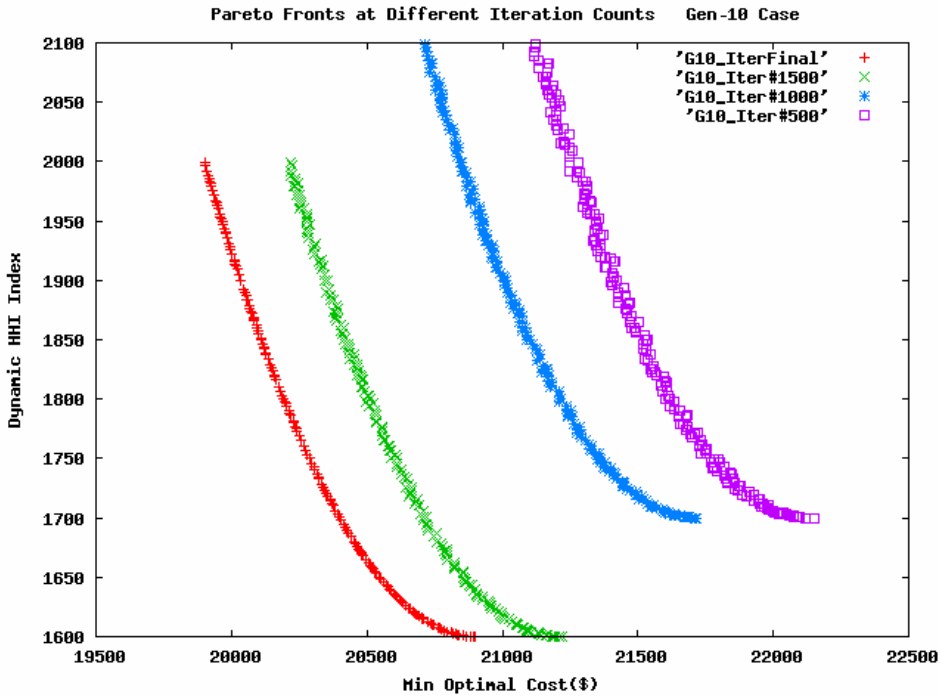


Figure 7.20 - Minimum Optimal Cost of Operation at each iteration using *NSGA II* Algorithm, 10-generator test case

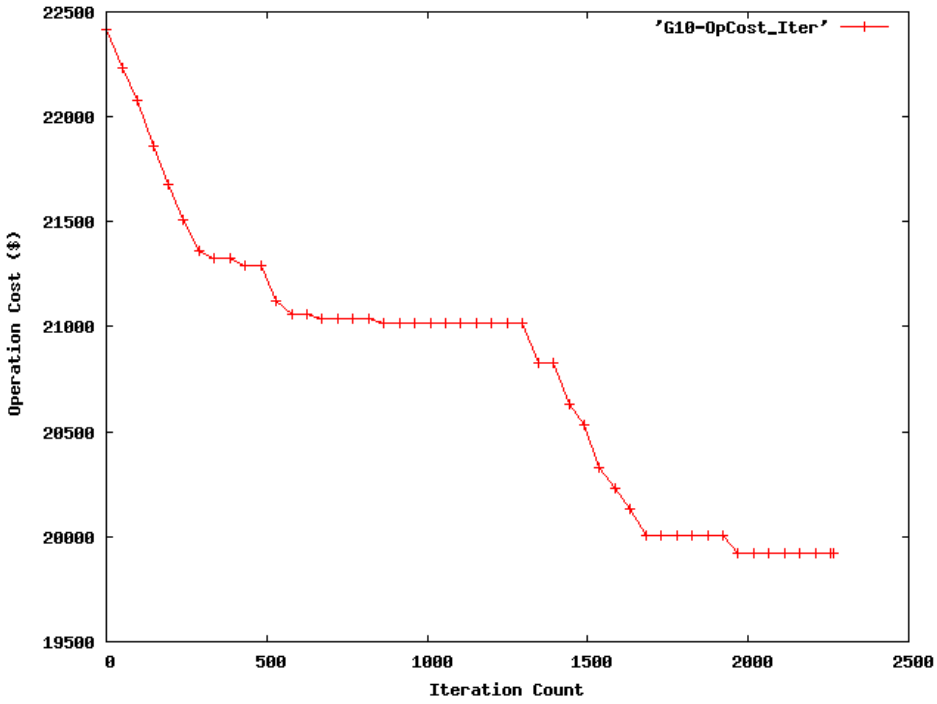
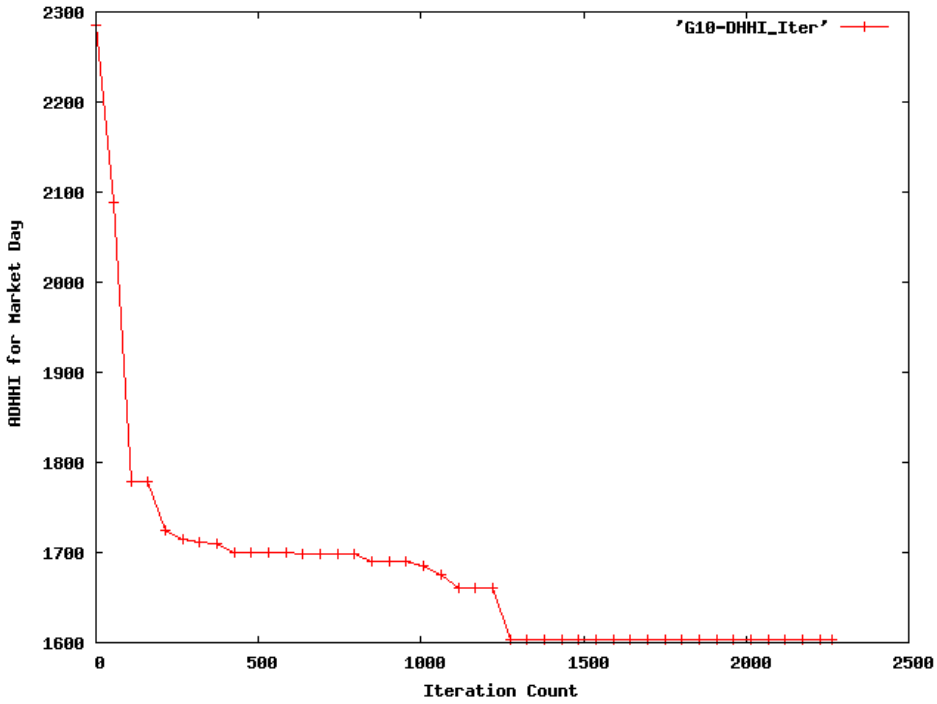


Figure 7.21 - Minimum *DHHI* at each iteration using *NSGA II* Algorithm, 10-generator test case



The next step is to select a single operating point at which the market will be dispatched based on the final pareto-front. Given that one of the key objectives is to ensure that the generator owners do not exhibit any market power, the industry-recommended moderate market-concentration-threshold value of 1800 for the *DHHI* was used as a guide to choose the suitable operating point from the pareto-front. From the pareto-front graph obtained for the *NSGA II* algorithm, a *DHHI* value of 1817 was found to be the closest to this threshold for the given power system and was chosen as the point at which the market was to be dispatched. The resulting generator-dispatch schedule for meeting the system load under this operating point is shown in Figure 7.22. From the results it is evident that the generators are chosen based on their economic-merit-order with many generators dispatched at their maximum economic values, while the dispatch of one generator was limited below its maximum economic dispatch value due to a line-overload-constraint preventing it from dispatching further. The rest of the generators are brought online to meet the load for those hours when the most economical generators cannot meet the demand. This is accomplished while minimizing the number of times a given generator is turned on and off during the 24-hour period. All dispatch schema recommended by each of the algorithms are reviewed to avoid any generator being turned on and off constantly within the 24 hour period. The approach taken here to minimize the number of times a given generator is turned on and off is by selecting the operating order of the generators for the entire market day and adjusting the output of each of the generators starting from the dispatch schema from the previous hour as part of the algorithm.

The resulting *DHHI* indices for generators 3 and 5 combined together is shown in Figure 7.23 for the selected operating point. The corresponding *DHHI* indices for each of the other 8 generators as well as for the entire system for every hour of the market day are also presented in the same figure for completeness. A tabular representation of the *DHHI* indices for the system for the market day chosen is shown in Table 7.6. The optimal final solution was chosen when the average *DHHI* for the entire system is just above 1800, and when the individual indices of generators 3 and 5 are also not above the recommended 1800. The overall average index for the entire system for the chosen market day was 1817.

Figure 7.22 - Generator match-up for the 10-generator test case

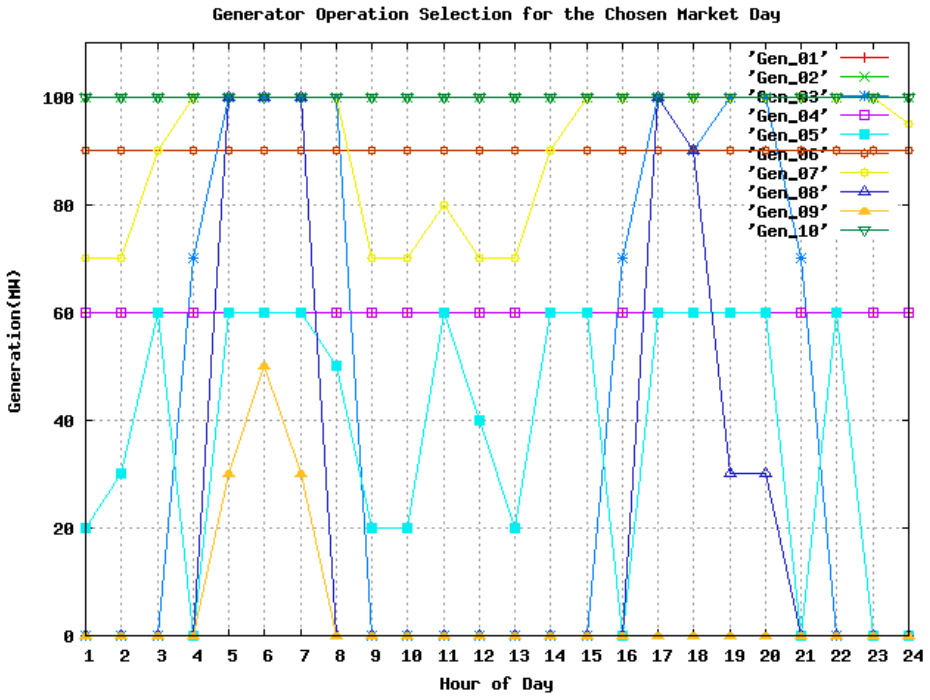


Figure 7.23 - *DHHI* charts for the 10-generator test case

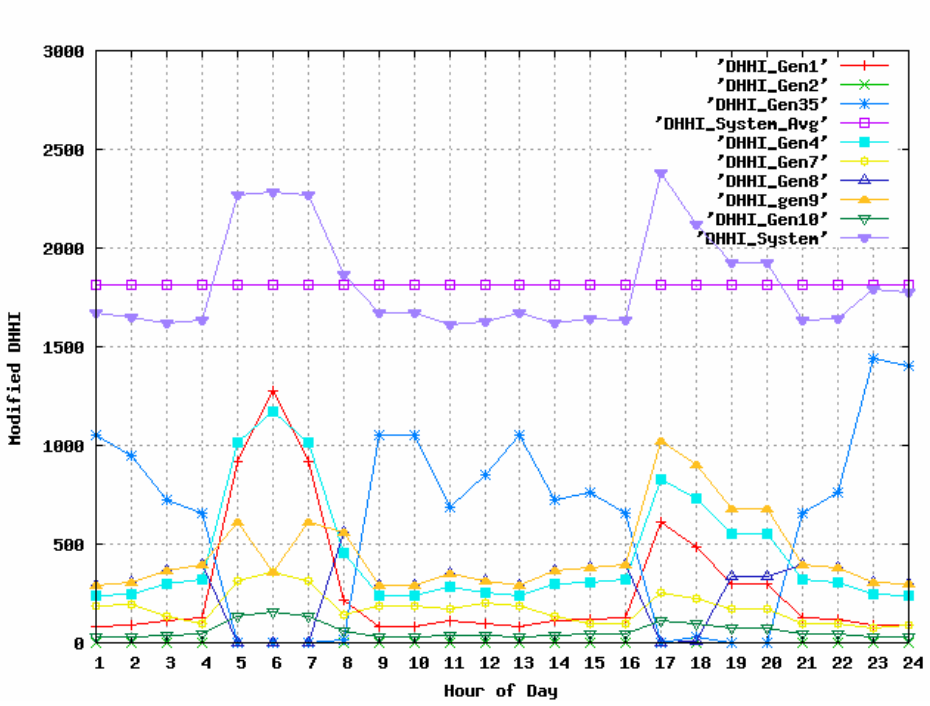


Table 7.6 - *DHHI* values for the System

Hour of Day	Gen 1	Gen 2	Gen 3& 5 Combined	Gen 4	Gen 6	Gen 7	Gen 8	Gen 9	Gen 10	Total System
1	86	0	1050	238	4	188	294	294	57	1663
2	90	0	948	246	4	194	304	304	59	1623
3	116	0	723	296	5	131	365	365	72	1643
4	129	0	653	320	5	98	395	395	79	1694
5	918	0	0	1011	19	312	0	611	302	2098
6	1275	0	0	1171	23	361	0	361	363	1921
7	918	0	0	1011	19	312	0	611	302	1663
8	216	0	14	452	7	139	558	558	117	1663
9	85	0	1050	238	4	188	294	294	57	1589
10	85	0	1050	238	4	188	294	294	57	1596
11	110	0	685	285	4	172	352	352	69	1663
12	94	0	848	285	4	201	315	315	62	1589
13	86	0	1050	238	4	188	294	294	57	1596
14	116	0	722	296	4	131	365	365	72	1663
15	122	0	763	307	5	94	379	379	76	1643
16	129	0	653	320	5	98	395	395	79	1711
17	610	0	0	826	5	255	0	1020	237	1694
18	484	0	30	730	15	225	9	901	204	2396
19	297	0	0	552	13	170	334	681	147	2092
20	297	0	0	552	9	170	334	681	147	1938
21	129	0	653	320	9	98	395	395	79	1694
22	122	0	763	307	5	94	379	379	76	1711
23	90	0	1437	246	4	76	304	304	59	1916
24	88	0	1404	242	4	90	299	299	58	1880

The results obtained from using the *LP* algorithm to solve the same market operation problem are also presented in Table 7.7 for comparison purposes. The total minimum operational cost using the *LP* algorithm, although lower than that of the other three methods, is seen to result in a *DHHI* beyond the desired values. The difference between the total minimum operation cost found from the *NSGA II* algorithm and the *LP* algorithm for this case is only $342.00. Since no effort to mitigate market power is undertaken in the *LP* algorithm, market manipulation activities through price inflations by the company owning generators 3 and 5 will go undetected, possibly creating a biased market. This clearly is an undesirable effect, given that the cost differentials between the results are insignificant for this case. It is also evident that the *LP* algorithm takes more than three times as long as the other three algorithms to find the optimal solution.

Table 7.7 - Comparison of results of algorithms for the 10-generator test case

Algorithm	# of Trials or Generations to convergence	CPU Time (Sec)	Total Minimum Operational Cost ($)	System *ADHHI*
MOTS	3126	7.34	20,571.00	1804
MOTS/GA	2113	6.01	20,256.00	1802
NSGA II	2265	5.85	20,202.00	1802
LP	8058	19.05	19,856.00	2460

In order to validate the robustness of an algorithm, and its ability to find the optimal solution with insensitivity to small changes in the starting solution, ten simulations on the same test system were conducted using the *NSGA II* algorithm. Here, the random seed input to the solution was changed from one test run to the next. As discussed previously, the effect of changing the initialization seed is to begin the iterations from a different starting solution set. From Table 7.8 it is evident that the random seed has a minimal impact on the final solution obtained. From the table it is also evident that the minimum operation costs in each test simulation are very close to one another. The largest of these minimum operating costs obtained in the repeated runs was $20,720.00, while the smallest was $20,182.00. Hence the difference between the best optimal cost and the worst optimal cost was only $538.00, which is less than a 3% variation.

Table 7.8 - Comparison of results between algorithms for the 10-generator test case

Random Seed Used	# of Generations to Converge	CPU Time (Sec.)	Total Minimum Operational Cost ($)	System *ADHHI*
0.1	2254	5.85	$20,656.00	1804
0.2	2251	5.98	$20,212.00	1801
0.3	2256	5.76	$20,720.00	1804
0.4	2256	5.68	$20,314.00	1803
0.5	2265	5.85	$20,202.00	1802
0.6	2256	5.77	$20,209.00	1803
0.7	2258	5.95	$20,182.00	1802
0.8	2256	5.68	$20,344.00	1801
0.9	2254	5.77	$20,468.00	1803
0.95	2253	5.93	$20,606.00	1800

7.6 Analysis of a 50-Generator, 20-Load power System with Congestion and Mitigated Market Power of Owners

In order to test the ability of each of the chosen algorithms to scale as the size of the problem grew, a power system comprising 50 generators and 20 loads was tested next. The test system used for this analysis is depicted in Figure 7.24. The corresponding linearized cost characteristics of the 50 generators are presented in Table 7.9. The intercept of the curve with the y-axis is presented as parameter "a" while the slope is represented as parameter "b" in the table. P_{max} and P_{min} values are the operation limits of a given generator. The load characteristics for the market day considered are presented in Table 7.10 and information on their buying offers is included in Appendix B.

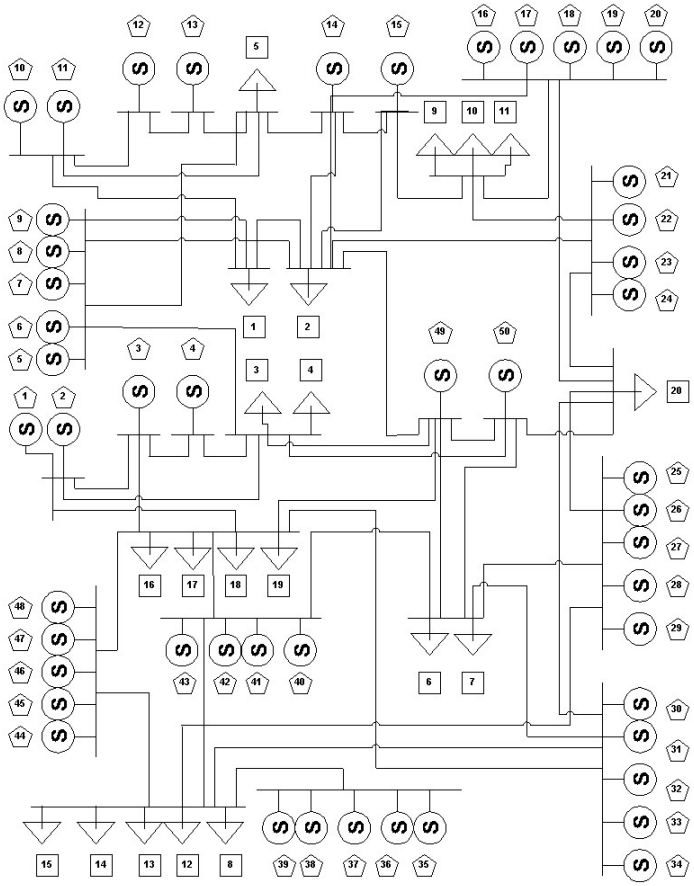
Table 7.9 - Cost Characteristics of 50 Generators

Gen #	a	b	Pmin	Pmax
1	9	0.45	20	160
2	15	0.75	20	300
3	30	0.6	50	265
4	16	0.8	20	60
5	15	0.75	20	250
6	30	1.5	20	400
7	25	0.5	50	500
8	66	2.2	30	400
9	22	1.1	20	100
10	90	1.8	50	250
11	60	3	20	60
12	15	0.75	20	300
13	19	0.95	20	200
14	12	0.6	20	60
15	35	1.75	20	300
16	60	3	20	90
17	75	2.5	30	300
18	48	2.4	20	350
19	8	0.4	20	100
20	6	0.3	20	60
21	10	0.5	20	60
22	10	0.5	20	100
23	10	0.2	50	400
24	18	0.9	20	60
25	15	0.75	20	60
26	8	0.4	20	90
27	6	0.3	20	150
28	46	2.3	20	50
29	15	0.3	50	500
30	8	0.4	20	60
31	24	1.2	20	60
32	10	0.5	20	300
33	18	0.9	20	100
34	22	1.1	20	60
35	22	1.1	20	60
36	22	1.1	20	400
37	10	0.5	20	100
38	12	0.6	20	50
39	6	0.3	20	100
40	4	0.2	20	60
41	6	0.3	20	60
42	10	0.5	20	100
43	8	0.4	20	100
44	16	0.8	20	150
45	10	0.5	20	60
46	6	0.3	20	90
47	4	0.2	20	100
48	10	0.5	20	50
49	18	0.9	20	100
50	14	0.7	20	60

Table 7.10 - Load Profiles

Hour	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
Total Load	5000	5680	5800	5850	6050	5950	5960	5000	5000	5140	5250	5000	5180	5600	5700	5800	6150	6110	6090	5500	5160	5000	5000	4500
Load_1	330	530	480	450	570	570	570	350	360	370	380	390	390	470	550	570	570	570	570	350	340	330	330	330
Load_2	150	150	150	170	160	160	160	160	180	180	200	200	200	180	180	180	180	180	160	160	160	160	130	125
Load_3	400	400	400	400	400	400	400	400	400	400	400	400	400	400	400	400	400	400	400	400	400	400	400	400
Load_4	280	280	280	280	280	230	230	220	200	200	200	200	200	200	230	250	250	260	260	260	280	280	250	250
Load_5	140	150	200	200	200	200	200	160	160	150	170	130	110	190	170	160	200	200	200	200	200	200	200	200
Load_6	340	350	400	400	400	400	400	360	360	350	370	330	310	390	370	360	400	400	400	400	400	400	400	400
Load_7	150	150	150	170	160	160	160	160	180	180	200	200	200	180	180	180	180	150	160	160	160	160	130	125
Load_8	400	400	400	400	400	400	400	400	400	400	400	400	400	400	400	400	400	400	400	400	400	400	400	400
Load_9	380	380	380	380	380	330	330	320	300	300	300	300	300	300	330	350	350	360	360	360	380	380	350	350
Load_10	140	150	200	200	200	200	200	160	160	150	170	130	110	190	170	160	200	200	200	200	200	200	200	200
Load_11	230	230	230	230	230	230	240	250	260	270	280	290	290	290	290	290	290	290	280	250	240	230	230	230
Load_12	150	150	150	170	160	160	160	160	180	180	200	200	200	180	180	180	180	150	160	160	160	160	130	125
Load_13	400	400	400	400	400	400	400	400	400	400	400	400	400	400	400	400	400	400	400	400	400	400	400	400
Load_14	380	380	380	380	380	330	330	320	300	300	300	300	300	300	330	350	350	360	360	360	380	380	350	340
Load_15	140	150	200	200	200	200	200	160	160	150	170	130	110	190	170	160	200	200	200	200	200	200	200	200
Load_16	220	500	570	570	570	570	570	280	260	430	340	270	550	570	570	570	570	570	570	420	0	0	0	0
Load_17	150	150	150	170	160	160	160	160	180	180	200	200	200	180	180	180	180	170	160	160	160	160	130	125
Load_18	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
Load_19	380	480	380	380	500	550	550	320	300	300	300	300	300	300	330	400	550	550	550	360	400	260	470	0
Load_20	140	200	200	200	200	200	200	160	160	150	170	130	110	190	170	160	200	200	200	200	200	200	200	200

Figure 7.24 - 50-Generator, 20-Load System



The effects of market concentration were investigated by considering generators 3, 5, 15, and 18 to be owned by a single company, while each of the other generators was considered to be owned by a different company. The resulting non-dominated pareto-optimal front graphs from each of the algorithms are depicted in Figure 7.25. The pareto-front from the *NSGA II* algorithm is observed to be closest to the origin, with results from *MOTS/GA* and *MOTS* algorithms being very close to one another along the fronts.

As done previously, the optimal point at which the market is to be operated was chosen using the industry-recommended *DHHI* of 1800 as a reference point. The corresponding *DHHI* that is closest to this reference value for the *NSGA II* solution is 1802. The lowest *DHHI* obtained from the *MOTS/GA* algorithm was 1806, while the corresponding value for *MOTS* algorithm was 1801. The optimal operating cost to operate the market at these market-power values is found to be $118,310.00 for *NSGA II*, $120,552.00 for the hybrid *MOTS/GA* and $120,630.00 for the *MOTS* algorithm. A few other performance-related metrics along with these operational details for each of the algorithms are presented in Table 7.11 for comparison.

Table 7.11 - Comparison of results between algorithms for the 50-generator test case

Algorithm	#Trials or Generations to Converge	CPU Time (Sec.)	Total Minimum Operational Cost ($)	*ADHHI*
MOTS	4598	18.50	120,630.00	1801
MOTS/GA	2130	14.50	120,552.00	1806
NSGA II	2456	12.50	118,310.00	1802
LP	13045	46.07	115,960.00	2640

From the results one can see that all three algorithms can be successfully used for solving the given market-dispatch problem while suppressing the exercise of market power by individual generator-owning companies. When comparing the results obtained for all three power systems, one could conclude that any of these algorithms can be successfully applied when the selected problem scales from a small 5-generator power system by a 10-fold increase to a 50-generator test case, as done in this simulation. As expected, the time for obtaining feasible non-dominant

solutions in each of the solution algorithms is seen to increase with the increase in scale of the problem. The total minimum operational cost found from the *LP* algorithm for this test case was $115,960.00, while the resulting average *DHHI* for the entire market day was 2640. This cost is $3,137.00 lower than what was found from the *NSGA II* algorithm. Considering the fact that the system has 50 generators, per-generator average difference is found to be approximately $62.75. That the observed average system-wide *DHHI* under the *LP* method falls into the high-market-concentration region, shows the merit of using a dual optimization method over a single optimization method.

The resulting *ADHHI* values found from the *NSGA II* algorithm for the entire system along with those for the company owning generators 3,5,15 and 18 are presented in Figure 7.26. As a comparison the corresponding *ADHHI* values for the three independent companies owning generators 7, 8 and 17, respectively, are also presented in the same figure. The fact that the market power of the company owning multiple generators has a high influence on the overall system market power is observable from these results. At an *ADHHI* of 1806 for the system, the *DHHI* values for the company owning multiple generators are seen to be contributing significantly to the overall system *DHHI* values. However, that the overall average *DHHI* value for the company with multiple generators is below the recommended threshold value is the guarantee that this company is not using its power under the selected operating point to manipulate the market.

Figure 7.25 - Pareto optimal front graphs from different algorithms for the 50- generator test case

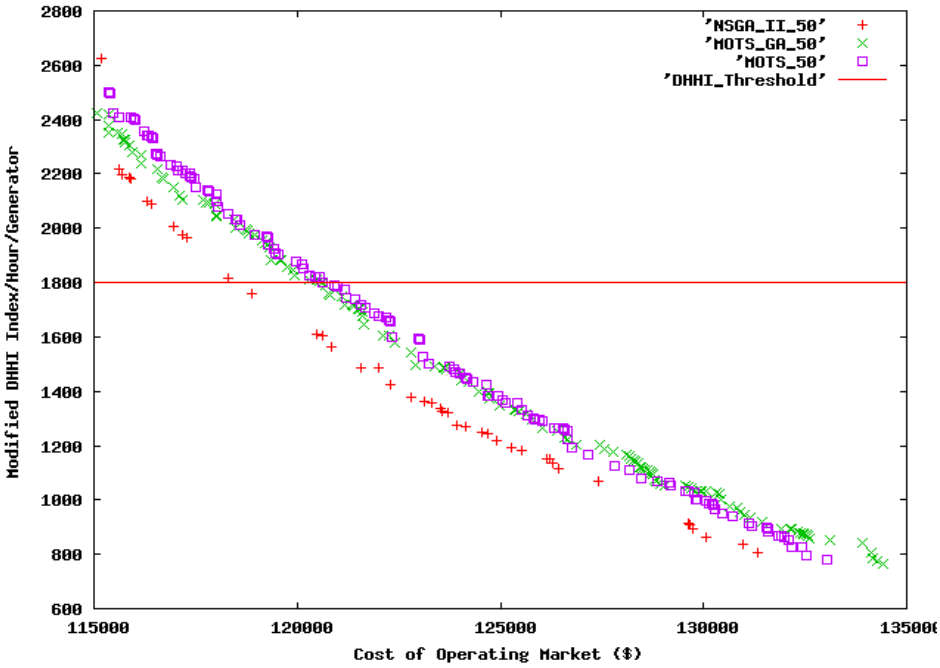
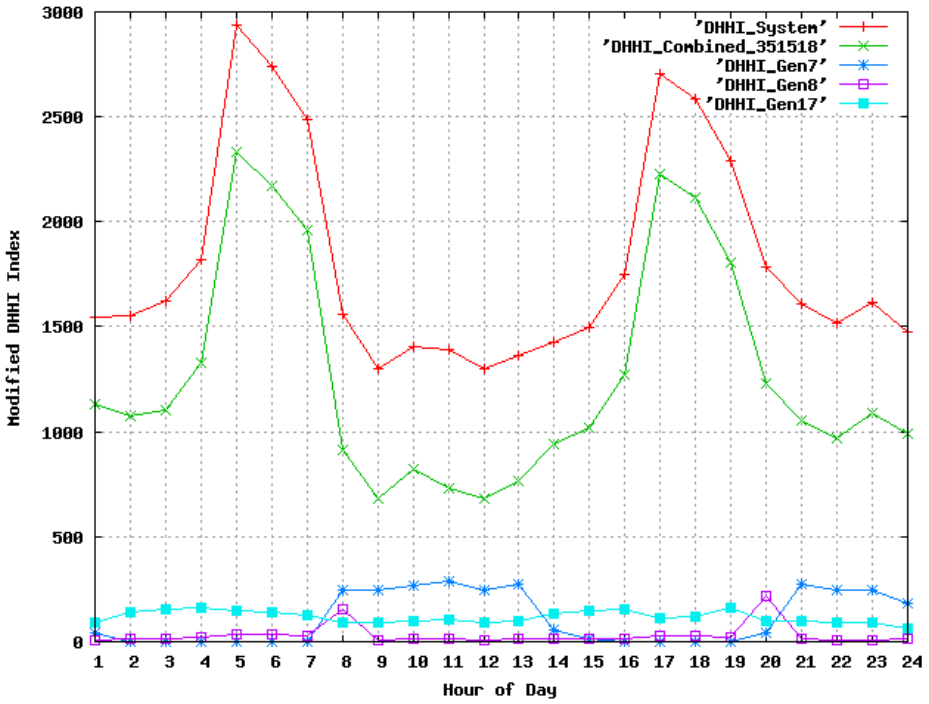


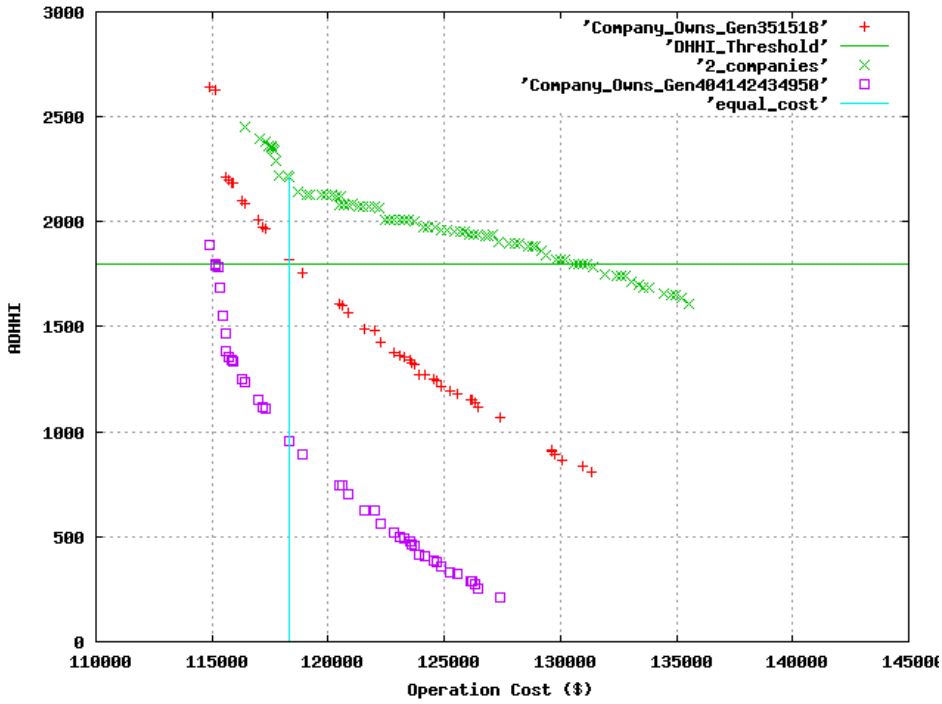
Figure 7.26 - Modified *DHHI* for the case where Gen 3, 5, 15, and 18 are owned by one company



The corresponding *ADHHI* values for the entire system along with those for the company owning the generators 3,5,15, and 18 are presented in the Table 7.12 below. The *ADHHI* values for every hour of the day for the remaining 46 generators are also presented in the table for completeness. From these results one can see that the potential of a given company to manipulate the market diminishes as the size of the market increases.

The effect when the generators owned by a single company are changed from the previously considered set of 3, 5, 15, and 18 to 40, 41, 42, 43, 49, and 50 while making generators 3, 5, 15 and 18 each be owned by a different company, without any other change to the system operating conditions, was considered next. In order to evaluate the effects of ownership change, the pareto-fronts obtained for each ownership arrangement were drawn in the same chart and are depicted in Figure 7. 27. As done previously, an operating point with a system *ADHHI* value of 1802 and an overall system operating cost of $118,310.00 was chosen. At this same operating cost the corresponding *ADHHI* value for the second ownership arrangement was found to be 894. If one were to move up the pareto-front for the second ownership arrangement, an alternative operating point with a lower economic operational cost could be selected. Using the same rationale as before, if the new operating point under this ownership arrangement was also selected around the industry-recommended 1800, a corresponding optimal operating cost of $115,460.00 with an *ADHHI* value of 1804 would be found. This demonstrates that a meager change of ownership or the strategic position of generators owned by a given company allows opportunities for market manipulation.

Figure 7.27 – Pareto-optimal Front Charts with Different Ownership Arrangements using *NSGA II* algorithm



Next, the impact of two companies owning multiple generators was considered at the previously selected operational point. Each other generator is considered to be owned by different company. For this analysis, company 1 was assumed to own generators 3, 5, 15, and 18 while company 2 was assumed to own generators 40, 41, 42, 43, 49, 50, respectively. With this arrangement, the resulting system *ADHHI* was found to be 2112 if we retain the overall cost of operating the market at $118,310.00. The fact that more companies owning more generators results in higher overall market concentration is evident from these results. Since this ownership arrangement is above the desired market-power threshold, a different operating point that would bring down the market power of the system while concurrently adjusting the market operating cost has to be found. If the system operator would move to the right of the corresponding pareto-optimal-front, an appropriate operating point where an *ADHHI* value is 1805, a value closer to the industry-recommended threshold value, could be found. At this value the total operating cost is found to be $131,858.00.

Table 7.12 - *DHHI* values for the 50 –Generator Power System with Generators 3, 5, 15, and 18 owned by a single company

Hour of Day	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
Total System	1522	1519	1589	1791	2943	2742	2482	1555	1283	1385	1372	1290	1350	1404	1470	1734	2711	2592	2285	1795	1608	1520	1621	1480
Combined	1132	1076	1102	1329	2336	2174	1962	912	683	820	729	683	767	939	1015	1274	2231	2117	1808	1230	1053	970	1090	994
Gen_1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Gen_2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Gen_4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Gen_6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Gen_7	39	0	0	0	0	0	0	245	245	268	288	245	275	59	16	0	0	0	0	40	271	245	245	183
Gen_8	10	16	17	18	37	34	31	157	10	11	12	10	11	15	16	17	28	26	23	220	11	10	10	16
Gen_9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Gen_10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Gen_11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Gen_12	10	16	17	18	37	34	31	10	10	11	12	10	11	15	16	17	28	26	23	14	11	10	10	7
Gen_13	39	63	70	31	37	34	31	10	39	43	46	39	44	59	64	63	28	32	23	14	18	39	39	29
Gen_14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Gen_16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Gen_17	88	142	157	164	148	138	124	88	88	96	104	88	99	134	145	157	111	117	165	100	98	88	88	66
Gen_19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Gen_20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Gen_21	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Gen_22	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Gen_23	157	142	157	164	333	310	279	88	157	96	141	157	99	134	145	157	251	238	203	124	98	88	88	117
Gen_24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Gen_25	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
Gen_26	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Gen_27	22	36	39	41	9	9	8	22	17	3	3	22	3	4	4	4	7	7	6	20	3	22	2	16
Gen_28	1	2	2	0	0	0	2	1	1	1	1	1	1	2	2	2	2	2	2	1	1	1	1	1
Gen_29	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Gen_30	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Gen_31	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Gen_32	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Gen_33	5	0	0	0	0	0	0	0	5	5	5	5	5	6	6	0	0	0	0	0	2	5	0	4
Gen_34	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Gen_35	1	1	0	2	0	0	0	0	1	1	0	0	1	0	0	2	0	0	0	0	2	0	2	1
Gen_36	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Gen_37	5	6	7	7	0	0	0	5	5	5	5	5	5	6	6	7	0	0	4	3	5	5	5	4
Gen_38	1	2	2	0	0	0	2	1	1	1	1	1	1	2	2	2	2	2	2	1	1	1	1	1
Gen_39	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Gen_40	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Gen_41	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Gen_42	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Gen_43	5	6	7	1	0	0	0	0	5	5	5	5	5	6	6	1	0	0	0	0	0	5	5	4
Gen_44	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Gen_45	1	1	0	2	0	0	0	0	1	1	0	0	1	0	0	2	0	0	0	0	2	0	2	1
Gen_46	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Gen_47	5	6	7	7	0	0	0	5	5	5	5	5	5	6	6	7	0	0	4	3	5	5	5	4
Gen_48	1	2	2	0	0	0	2	1	1	1	1	1	1	2	2	2	2	2	2	1	1	1	1	1
Gen_49	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Gen_50	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Repeated simulations for the power system with the single company owning the group of generators 40, 41, 42, 43, 49, 50 using the *NSGA II* algorithm with different initialization seeds were conducted to ensure that the initial solution set does not have any impact on the algorithms' ability to arrive at the final pareto-solution-set. Results from 10 repeated simulations demonstrate the robustness in the solution algorithm. From the results it is evident that the difference between the highest and lowest minimum operational costs is $1435.00. When compared with the overall cost of operating the market, this is seen to be less than 2%. The results from these repeated simulations are presented in Table 7.13 below.

Table 7.13 - Comparison of results of changing initial random seed using the *NSGA II* algorithm for the 50-generator test case

Random Seed Used	# of Generations to Converge	CPU Time (Sec.)	Total Minimum Operational Cost ($)	Modified HHI/Hour
0.1	2559	12.80	115,156.00	1803
0.2	2556	12.50	115,460.00	1804
0.3	2554	12.40	115,490.00	1803
0.4	2562	13.10	116,425.00	1802
0.5	2550	12.30	115,650.00	1805
0.6	2557	12.60	114,990.00	1806
0.7	2551	12.35	115,750.00	1801
0.8	2548	12.20	115, 512.00	1802
0.9	2556	12.55	115,248.00	1804
0.95	2564	13.20	115, 765.00	1806

7.7 Analysis of a 50-Generator 20-Load Power System with Congestion and Mitigated Market Power between Generators with consolidation of generator ownership

The next test case considered was to evaluate the impact of ownership consolidation on the market. For this the same test power system was used, keeping all other constraints and conditions the same, except the number of generators owned by a given company. Here, it was assumed that only one company was involved in acquiring more and more generators from one situation to the other, while each of the remaining generators was owned by a different company. The study began with the case where the given company owned 5 generators. The ownership was increased by 5 more generators at each step. The lowest *ADHHI* value from each pareto-front resulting from each change of ownership is shown in Figure 7.28. As expected, the *ADHHI* value goes beyond acceptable levels when the number of generators owned by the given company increases beyond 20. Groupings of generators owned by the same company for each scenario considered are tabulated below.

Table 7.14 - Generator ownership by the company dominating the Market; 50- test case 4

Number of Generators owned by company	Generator identifier
5	3,5,15,18,23
10	3,5,15,18,23,40,41,42,43,49
15	3,5,15,18,23,40,41,42,43,49,50,32,33,34,35
20	3,5,15,18,23,40,41,42,43,49,50,32,33,34,35,14,16,20,21,22
25	3,5,15,18,23,40,41,42,43,49,50,32,33,34,35,14,16,20,21,22,8,9,10,11,12
30	3,5,15,18,23,40,41,42,43,49,50,32,33,34,35,14,16,20,21,22,8,9,10,11,12, 27,28,29,30,31
35	3,5,15,18,23,40,41,42,43,49,50,32,33,34,35,14,16,20,21,22,8,9,10,11,12, 27,28,29,30,31,1,2,4,6,7
40	3,5,15,18,23,40,41,42,43,49,50,32,33,34,35,14,16,20,21,22,8,9,10,11,12, 27,28,29,30,31,1,2,4,6,7,13,36,37,38,39
45	3,5,15,18,23,40,41,42,43,49,50,32,33,34,35,14,16,20,21,22,8,9,10,11,12, 27,28,29,30,31,1,2,4,6,7,13,36,37,38,39,44,45,46,47,48
50	All 50 generators

The results from this study are presented in Table 7.15 below. From the results one can observe that as the number of generators owned by the company increases, the lowest *ADHHI* achievable is always higher than the industry-recommended threshold value of 1800. The corresponding pareto-optimal fronts are also depicted in Figure 7.29. It is evident that as the ownership increases the point with the lowest possible *ADHHI* is selected as the appropriate operational point since the industry-recommended index value cannot be achieved.

Table 7.15 - Results from Consolidation of Generator Ownership

Number of Generators Owned by the Consolidated Company	System-wide *ADHHI*
5	1802
10	1805
15	1812
20	1817
25	3447
30	4012
35	5829
40	7217
45	8373
50	10000

Figure 7.28 - *ADHHI* value with different generator ownership using *NSGA II* algorithm; 50- generator test case 4

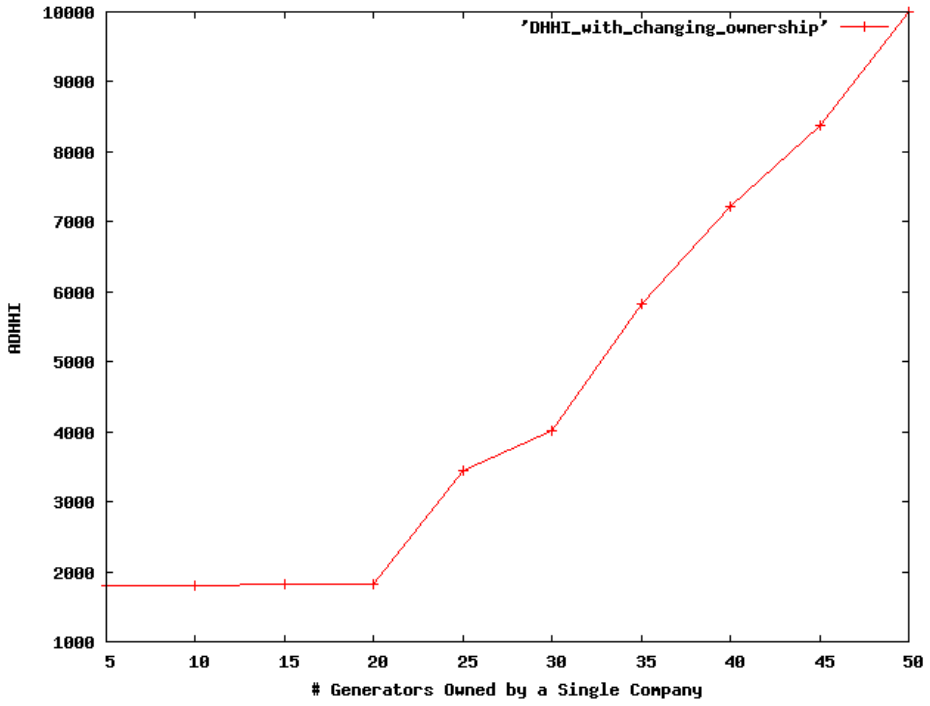
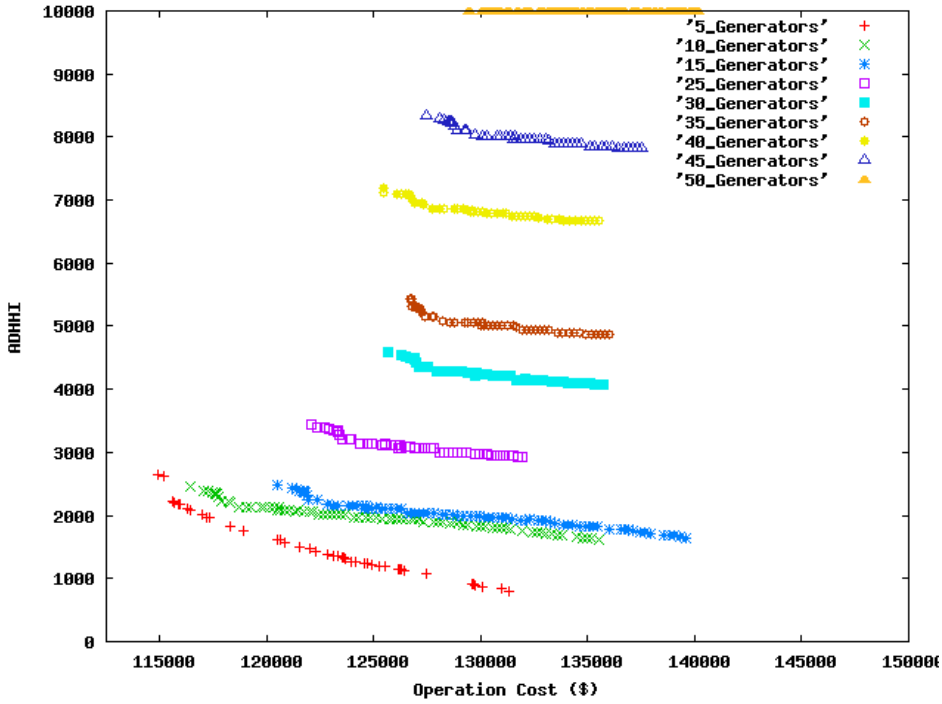


Figure 7.29 – Pareto-Optimal Front Charts with different generator ownership using *NSGA II* algorithm; 50- generator test case 4



7.8 Optimal Operational Point Selection from a Pareto-front for a 50-Generator 20-Load Power System

Although the Department of Justice has suggested a prescriptive set of market-concentration ranges, the use of these ranges to make operational decisions without evaluating their reasonableness in a given problem seems too naive. In all previous analysis in this thesis, a *DHHI* value of 1800, which is the upper limit for moderate market concentration, was chosen as the suitable operating point from the *pareto-optimal-solution-set* found by the multi-objective optimization. A more justifiable selection criteria as used by [27], [28], based on fuzzy optimality definitions, as presented in section 6.5 of this dissertation was next evaluated for the day-ahead market dispatch problem.

The reasonability of using a threshold-based average *DHHI* for the entire system as used in the previous analysis was investigated. For this the average system-wide *DHHI* along with the peak system-wide *DHHI* for the market day considered, average *DHHI* for the company that has the highest market power, and the peak *DHHI* for that company were compared side by side with the overall optimal economic operational cost of the market. Given that the company which owns generators 3, 5, 15, and 18 has a higher market-power than the company which owns generators 40,41,42,43,49 and 50, the corresponding values for the first company were compared with the system-wide values. The results obtained from the *NSGA II* algorithm were used in the evaluations. Table 7.16 shows the results from this analysis.

Table 7.16 - Candidate Solution Set Considered for Multi-Criteria Decision Making

Total Operational Cost ($)	System-wide *ADHHI*	System-wide Peak *DHHI*	*ADHHI* for a company owing multiple generators	Peak *DHHI* for a company owing multiple generators
130811.90	1924	4221	1371	3360
131092.10	1914	4212	1363	3352
131132.30	1913	4105	1380	3278
131232.30	1913	3945	1375	3148
131309.00	1899	3567	1338	2880
131309.00	1899	3563	1390	2859
131310.90	1899	3561	1406	2867
131318.70	1873	3496	1361	2800
131342.00	1855	3421	1377	2757
131595.80	1828	3356	1338	2693
131595.80	1828	3265	1306	2601
131604.40	1827	3179	1290	2573
131802.10	1808	3087	1271	2496
131835.70	1805	2936	1267	2335
131839.00	1804	2932	1301	2343
131840.00	1803	2933	1340	2369
131875.40	1784	2857	1325	2307
132238.80	1750	2824	1253	2251
132560.00	1743	2812	1226	2277
132561.90	1743	2809	1227	2276

Each of the solutions was compared with the other 19 from the above table. Each time a competing solution is found less desirable on any one of the five objectives being evaluated when compared with the candidate solution being considered, the rank of the corresponding comparison cell of the matrix shown below is updated so that the number of objectives in which

a given candidate solution is better than another is reflected. As an example, if the first candidate solution is compared with the 3rd candidate solution, one can see that out of the five objectives, the total operational cost and average *DHHI* for the company owning multiple generators are better in the 1st candidate solution than in the 3rd, resulting in a ranking of 2 for the cell [1,3] of the matrix below. Similarly one can clearly see that cell [3,1] will be 3 to ensure that the sum of [1,3] and [3,1] will always be equal to 5, which are total number of criteria being considered. This procedure was repeated to fill out the entire matrix shown below.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	0	1	2	2	1	2	2	1	2	1	1	1	1	1	1	1	1	1	1	1
2	4	0	2	2	1	2	2	1	2	1	1	1	1	1	1	1	1	1	1	1
3	3	3	0	2	1	2	2	1	1	1	1	1	1	1	1	1	1	1	1	1
4	3	3	3	0	1	2	2	1	2	1	1	1	1	1	1	1	1	1	1	1
5	4	4	4	4	0	3	3	2	2	1	1	2	1	1	1	1	1	1	1	1
6	3	3	3	3	2	0	4	1	1	1	1	1	1	1	1	1	1	1	1	1
7	3	3	3	3	2	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1
8	4	4	4	4	3	4	4	0	2	1	1	1	1	1	1	1	1	1	1	1
9	3	3	4	3	3	4	4	3	0	1	1	1	1	1	1	1	1	1	1	1
10	4	4	4	4	4	4	4	4	4	0	2	1	1	1	1	2	1	1	1	1
11	4	4	4	4	4	4	4	4	4	3	0	1	1	1	1	2	2	1	1	1
12	4	4	4	4	4	4	4	4	4	4	4	0	1	1	2	2	2	1	1	1
13	4	4	4	4	4	4	4	4	4	4	4	4	0	1	2	2	2	1	1	1
14	4	4	4	4	4	4	4	4	4	4	4	4	4	0	3	2	2	1	1	1
15	4	4	4	4	4	4	4	4	4	4	4	3	3	2	0	1	2	1	1	1
16	4	4	4	4	4	4	4	4	4	3	3	3	3	2	1	0	1	1	1	1
17	4	4	4	4	4	4	4	4	4	4	3	3	3	3	3	4	0	2	2	2
18	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	0	2	2
19	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	3	0	2
20	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	3	3	0

The next step is to retain only those solutions that are k_f -dominant based on fuzzy multi-decision criteria. Using a dominance factor of 0.25, one could eliminate all solutions where the ranking is either 1 or 4. This results in the reduced matrix as shown below and it is clear that only the solutions 2,3,4 and 6 are left from the starting set of 20 candidate solutions. From this reduced matrix, it is evident that candidate solution 6 is better than all three of the other solutions and will be selected as the optimal solution from the candidate pareto-set of solutions.

	2	3	4	6
2	0	2	2	2
3	3	0	2	2
4	3	3	0	2
6	3	3	3	0

At this optimal solution, the cost of operating the day-ahead market is $131,309.00 with a system-wide *ADHHI* of 1899. The corresponding peak of system-wide *DHHI* is found to be 3563, while the *ADHHI* for the company with the highest market power is found to be 1390. The peak *DHHI* for the same company is found to be 2859 at this optimal solution. If this solution is compared with the cost of operating the market around the industry-recommended *ADHHI* index of 1803, with a total operating cost of $131,840.00, one will observe an overall reduction of $531.00 for operating the market. However, the resulting *ADHHI* has increased by 96, while the peak of system-wide *DHHI* has increased by 230. The corresponding increase in *ADHHI* for the company with the highest market power is 50, while there is an overall reduction of 490 in the peak *DHHI* for the same company. From these comparisons one could conclude that the solution found from the multi-decision-making criteria is better in many regards: it lowered the total cost of operating the market, while reducing the peak *DHHI* for the company that has the highest potential to influence the market. At this operating scheme, the total cost of operating the market over a 30-day period would realize a reduction of $15,930.00 or 12% in total operating costs.

Although a net positive optimal operating cost is found for each of the cases studied in this research, a day-ahead market operated by ISO/RTO needs to ensure that there is no revenue imbalance. Therefore, the optimal operational cost found in each case will have to be socialized based on the load share of each load in the system in addition to the price they are willing to pay to the market. This process will ensure the cost of operating all generators and the power system is recovered. The revenue imbalance is socialized using a pro-rata share to each load based on its consumption in each hour of the market day. This revenue allocation is commonly referred to as revenue sufficiency guarantee (RSG) distribution process in market operations.

CHAPTER 8 - Conclusions

The work in this dissertation demonstrates the applicability of three promising non-traditional optimization algorithms for solving a multi-objective, day-ahead, market-clearing problem. Given that a majority of transactions in an electricity market take place in its day-ahead market, optimal operation of the day-ahead market would yield definite benefits to all market participants. The algorithms tested and the methodologies used offer a coherent and a consistent framework for analyzing different operational scenarios of a given day-ahead market, including conditions when the power system is constrained as well as when companies resort to market manipulation.

From the results, it is evident that all three algorithms perform satisfactorily for the problem being solved. However, it is also observed that each of these algorithms has its own merits and demerits. Looking closely at the *MOTS* algorithm, one can see that one of the biggest challenges in the method is the selection of a good initial solution. This is because the method heavily relies on searching around a given neighborhood of solutions. It is also evident that the selection of other key parameters such as the *tabu list length* plays a vital role in overall solution quality and the time it takes for finding the optimal global solution. Similarly, the combined *MOTS/GA* algorithm, although not so heavily dependent on its initial solution, has its own challenges in selecting key parameters such as the population size, generation size and other pertinent parameters. Of the three algorithms investigated, the *NSGA II* algorithm appears to have the least sensitivity to changes in the configuration parameters.

Comparing the results between the 3 test systems one can clearly see that all three algorithms tested have the ability to scale up as the size and complexity of problem scales up. The results from many test cases also show that the times for solving the given market-dispatch problem with a secondary objective of operating at a minimal market-power increase exponentially as the size of the power system increases. From the results one could conclude that any one of the tested algorithms could be interchangeably used by a regional transmission organization that operates a day-ahead market. Based on factors such as ease of use, speed of

solving and solution quality, one can lean towards using the *NSGA II* algorithm, with the combined *MOTS/GA* algorithm following behind. It is worthwhile to note that although the *MOTS* algorithm appears the least favorite when compared with the other two algorithms based on the solution quality and performance, it has the advantage of not needing to model the problem using chromosomes. Work done previously using genetic algorithms indicates that the solution time has a key dependence on the length of the chromosome that is modeled to represent a given problem, especially when the problem-space becomes large. Given that both *NSGA II* and *MOTS/GA* methods were able to find the *pareto-fronts* successfully, one could conclude that the chromosome design approach that was used in this study was appropriate. Impact of chromosome modeling when applied to a large system is a possible proposal for future research.

The fact that the operational solutions recommended by the *LP* algorithm resulted in high market concentrations clearly demonstrate the challenges currently faced by market operators. This is due to the fact that a two-step approach to mitigating the effects of market manipulations is currently used.

One strength of this study is its use of multi-objective solutions that optimize two competing objective functions simultaneously when evaluating the market operations. Since all three algorithms are multi-objective optimization algorithms, they can be easily extended to include more than two objective functions. Measures of lost opportunity and environmental impact are some possible dimensions that can be easily incorporated into future analyses.

The first set of evaluations considered the market to be operated at an industry recommended threshold value. When the results from each algorithm were compared at this threshold value, one could see that the *NSGA II* algorithm offers the best solution for most of the scenarios studied.

Since almost all market regions have to deal with their neighbors, interactions from neighboring regions in terms of buying from the market, selling into the market and moving power through the market region need to be considered. The influences of neighboring regions on the day-ahead market are yet another extension that can be included in future research.

From the results, one can conclude that all three algorithms investigated are suitable for solving the given day-ahead-market-dispatch problem. The introduction of fuzzy decision methods enriched the capabilities of the solution approaches investigated by providing additional insights to selecting an operational point where each of the considered objective functions is not dominating any other.

Bibliography

1. F.C. Schweppes, M. Caramanis, R. Taboras and R. Bohn, *Spot Pricing of Electricity*, Kluwer Academic Publishers, Boston 1988.
2. J. Ma, Y.H. Song, Q. Lu, and S. Mei, "Framework for dynamic congestion management in open power markets**",** *IEE Proceedings on Generation, Transmission and Distribution,* Volume 149, Issue 2, March 2002 Page(s):157 - 164.
3. A.K. David, "Dispatch Methodologies for Open Access Transmission Systems", *IEEE Transactions on Power Systems*, Volume 13, No. 1, February 1998 Page(s): 46-53.
4. C. J. Day, B. F. Hobbs, J. Pang, "Oligopolistic Competition in Power Networks: A Conjectured Supply Function Approach*"*, *IEEE Transactions on Power Systems*, Volume 17, No. 3, August 2001 Page(s): 597-607.
5. A.J. Conejo, J. Contreras, J. M. Arroyo, S. de la Torre, "Optimal Response of an Oligopolistic Generating Company to a Competitive Pool-Based Electric Power Market", *IEEE Transactions on Power Systems*, Volume 17, No. 2, May 2002 Page(s): 424-430.
6. A.R. Kian, J. B. Cruz, Jr. "Nash Strategies for Load Serving Entities in Dynamic Energy Multi Markets", *Proceedings for the Hawaii International Conference on System Sciences*, January 7-10, 2002 Page(s): 730-738.
7. H. Song, C. Liu, and J. Lawarree, "Nash Equilibrium Bidding Strategies in a Bilateral Electricity Market", *IEEE Transactions on Power Systems*, Volume 17, No.1 , February 2002 Page(s): 73-79.
8. J. Yao, B. Williams, S. Oren, "Cournot Equilibrium in Price Capped Two Settlement Electricity Markets", *Proceedings of the 38th Hawaii International Conference on System Sciences*, 2005 Page(s): 58c-58e.
9. J. Yang, G. Jordan, "System Dynamic Index for Market Power Mitigation in the Restructuring Electricity Industry", *IEEE Society Summer Meeting,* Volume 4,16-20 July 2000 Page(s):2217 – 2222.
10. J.W. Bailek, "*Gaming* the Uniform-Price Spot market: Quantitative Analysis", *IEEE Transactions on Power Systems*, Volume 17, No.3, August 2002 Page(s): 768-773.

11. F.L. Alvarado, "Market Power: A Dynamic Definition", *Conference on Bulk Power System Dynamics and Control – IV Restructuring*, Santorini, Greece, August 24-28, 1998 Page(s):1-4.
12. R.A.S.K. Ranatunge, U.D. Annakkage, C.S. Kimble, "Linear Programming based Algorithm for Reactive Power Constrained Real Power Dispatch and Pricing", LESCOPE '01, 2001 *Conference on Large Engineering Systems in Power Engineering*, July, 11-13, July 2001 Page(s): 2-6.
13. S. Dekrajangpetch, G. Sheble, "Auction Implementation Problem using LaGrangian Relaxation", *IEEE Transactions on Power Systems*, Volume 14, No 1, February 1999 Page(s): 82-88.
14. L. F. Sugianto, M Widjaja, "Optimizing Bidding Strategy in the Australian National Electricity Market", *International Journal of Fuzzy Systems*, Volume 3, December 2-5, 2001 Page(s): 532-540..
15. L. Motto, and F.D. Galliana, "Equilibrium of Action Markets with Unit Commitment: The Need for Augmented Pricing", *IEEE Transactions on Power Systems*, Volume 17, No. 3, August 2002 Page(s): 798-805.
16. A. Baykasoglu, L. Özbakir, A.I. Sönmez, "Using Multiple Objective Tabu Search and Grammars to model and solve multi-objective flexible Job Shop Scheduling", *Journal of Intelligent Manufacturing,* December 2004, Page(s): 777-785.
17. A. G. Bakirtzis, P. N. Biskas, C. E. Zoumas, V. Ptetridis, "Optimal Power Flow by Enhanced Genetic Algorithm", *IEEE Transactions on Power Systems*, Volume 17, No. 2, May 2002 Page(s):229-236.
18. R.T.F.A. King, and H.C.S. Rughooputh, "Elitist Multiobjective Evolutionary Algorithm for Environmental/Economic Dispatch", *The 2003 Congress on Evolutionary Computation*, Volume 2, December, 8-12, 2003. Page(s):1108-1114.
19. K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, "A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II", *IEEE Transactions on Evolutionary Computation*, Volume 6, No. 2, April 2002 Page(s): 182-197.
20. K.P. Dahal, S.J. Galloway, G.M. Burt, J.R. McDonald, "Generation Scheduling using Genetic Algorithm based Hybrid Techinques", LESCOPE '01, 2001 *Conference on*

Large Engineering Systems in Power Engineering, July, 11-13, July 2001 Page(s): 74-78.

21. W. Ongsakal, N. Ruangpayoongsak, "Constrained Dynamic Economic Dispatch by Simulated Annealing/ Genetic Algorithms" , *22nd IEEE Power Engineering Society International Conference on Power Industry Computer Applications*, May 20-24, 2001 Page(s): 207-212.
22. M. Zdansky, J. Pozivil, "Combination Genetic/Tabu search Algorithm for Hybrid Flowship Optimization", *Proceedings of Algoritmy 2002* ,16th Conference on Scientific Computing, Vysoke Tatry –Podbanske, Sloakia, September 8-13, 2002 Page(s): 230-236.
23. M. Madrigal, V. H. Quintana, "Optimal Day-ahead Network Constrained Power System's Market Operations Planning Using and Interior Point Method*", IEEE Transactions on Power Systems,* Volume 14, No. 1, February 1998 Page(s):401-404.
24. G. Hammond, I. Bradley, "Assessment of Transmission Congestion Cost and Locational Marginal Pricing in a Competitive Electricity Market", *IEEE Transactions on Power Systems*, Volume 19, No. 2, May 2004 Page(s): 769-775.
25. I. J. Ramirez-Rosado, J. A. Dominguez-Navarro, "New Multi-Objective Tabu Search Algorithm for Fuzzy Optimal Planning of Power Distribution Systems", *IEEE Transactions on Power Systems*, Volume 21, No. 1, Feb. 2006 Page(s):224 – 233.
26. A. H. Mantawy, Y.L. Abdel-Magid, S. Z. Selim, "Integrating Genetic Algorithms, Tabu Search, and Simulated Annealing for the Unit Commitment Problem", *IEEE Transactions on Power Systems*, Volume 14, No. 3, Aug. 1999 Page(s): 829 – 836.
27. R.R. Yager, "On Ordered Weighted Averaging Aggregation Operators in Multi-Criteria Decision Making", *IEEE Transactions on Systems Man and Cybernetics*, Volume 18, January/February 1988 Page(s): 183-190.
28. F. Herrera, E. Herrera-Viedma, and J.L. Verdegay, "On Dominance Degrees in Group Decision Making with Linguistic Preferences", *Proceedings Current Issues Fuzzy Technologies*, Trento, Italy, June 1-3, 1994 Page(s): 113-117.
29. M. Farina and P. Amanto, "A Fuzzy Definition of Optimality for Many- Criteria Optimization Problems", *IEEE Transactions on Systems Man and Cybernetics*, Volume 34, No. 3, May 2004 Page(s):315-326.

30. M. Farina and P. Amanto, “On the Optimal Solution Definition for Multi-Criteria Optimization Problems”, *Proceedings of the NAFIPS-FLINT International Conference*, IEEE Service Center, Piscataway, New Jersey, June 2002 Page(s): 233-238.
31. M. Koppen, R. Vicente-Garcia, and B. Nickolay, “Fuzzy-Pareto-Dominance and its Application in Evolutionary Multi-Objective Optimization”, *Evolutionary Multi-Criterion Optimization*, Third International Conference, Guanajuato, Mexico, March 2005 Page(s): 399-412.
32. Restructured Electricity Markets: California Market Desgin Enabled Excercise of Market Power, Report to Congressional Requestors for the United States General accounting Office, June 2002. Page(s): 1-47.
33. J. Yang, G., Jordan, “System Dynamic Index for Marekt Power Mitigation in the Restructuring Electricity Industry, *IEEE Power Engineering Socieity Meeting*, Volume 4, No. 4, 2000 Page(s): 2217-2222.
34. G. Huang, K. Song, “A Simple Two Stage Optimization Algorithm for Constrained Power Economic Dispatch”, *IEEE Transactions on Power Systems*, Volume 9, No. 4, November 1994 Page(s): 1818-1824.
35. A.F. Rahimi, A.Y. Sheffrin,”Effective Market Montoring in Deregulated Electricity Markets”, *IEEE Transactions on Power Systems*, Volume 18, No 2, May 2003 Page(s): 486-493.
36. J. Yao, B. Williams, S. Oren, “Cournot Equilibrium in Price Capped Two Settlement Electricity Markets”, *Proceedings of the 38th Hawaii International Conference on System Sciences*, 2005 Page(s) 1-10.
37. A. Bayasoglu, “A MultiObjective Tabu Serach based Simulation Optimization Approach for Loading of Cellular Manufacturing Systems”, *Industrial Engineering*, 12(1), 2001 Page(s): 2-24.
38. A.J. Svoboda, S.S Oren, “Integrating Price-based Resources in Short-Term Scheduling of Electric Power Systems”, *IEEE Transactions on Energy Conversion*, Volume 9, No. 4, Dec. 1994. Page(s): 760 – 769.
39. C.C. Rajan, M.R. Mohan, “An Evolutionary Programming-based Tabu Search Method for Solving the Unit Commitment Problem”, *IEEE Transactions on Power Systems*, Volume 19, No. 1, Feb. 2004 Page(s): 577 – 585.

40. A.T. Bryant, D.M. Jaeggi, G.T. Parks, P.R. Palmer, "The Influence of Operating Conditions on Multi-Objective Optimization of Power Electronic Devices and Circuits", *Industry Applications Conference*, 2005. Fortieth IAS Annual Meeting. Conference Record of the 2005, Volume 2, 2-6 Oct. 2005 Page(s):1449 – 1456.
41. M. A. Plazas, A. J. Conejo, F. J. Prieto, "Multimarket Optimal Bidding for a Power Producer", *IEEE Transactions on Power Systems*, Volume 20, No. 4, Nov. 2005 Page(s): 2041 – 2050.
42. A. Kanagala, M. Sahni, S. Sharma, B. Gou, J. Yu, "A Probabilistic Approach of Hirschman-Herfindahl Index (HHI) to Determine Possibility of Market Power Acquisition", *IEEE PES Power Systems Conference and Exposition*, Volume 3, Oct. 2004 Page(s): 1277 - 1282.
43. J. Tippayachai, W. Ongsakul, I. Ngamroo, "Nonconvex Economic Dispatch by Enhanced Tabu Search Algorithm", *IEEE Power Engineering Society General Meeting*, Volume 2, 13-17 July 2003 Page(s):908-913.
44. Z. Li, Z. Jianguo, H. Xueshan, N. Lin, "Day-Ahead Generation Scheduling with Demand Response", *IEEE/PES Transmission and Distribution Conference and Exhibition*: Asia and Pacific, 2005, 15-18 Aug. 2005 Page(s): 1 – 4.
45. R.C. Garcia, J. Contreras, M. van Akkeren, J. B. C. Garcia, "A GARCH Forecasting Model to Predict Day-Ahead Electricity Prices", *IEEE Transactions on Power Systems*, Volume 20, No. 2, May 2005 Page(s):867 – 874.
46. F. Yong, M. Shahidehpour, L. Zuyi, "Security-Constrained Unit Commitment with AC Constraints", *IEEE Transactions on Power Systems*, Volume 20, No 2, May 2005 Page(s): 1001 – 1013.
47. C. Chen, J. Cruz, Jr., "Stackelburg Solution for Two-Person Games with Biased Information Patterns", *IEEE Transactions on Automatic Control*, Volume 17, No. 6, Dec 1972 Page(s):791 – 798.
48. T. Ray, K Tai, C. Seow, "An Evolutionary Algorithm for Multi-Objective Optimization", *Engineering Optimization*, Volume 33, No 3, 2001 Page(s): 339-424.
49. Milosevic, B.; Begovic, M, "Nondominated Sorting Genetic Algorithm for Optimal Phasor Measurement Placement*", IEEE Transactions on Power Systems*, Volume 18, No. 1, 2003 Page(s): 69-75.

50. Michael E. Agnes, Webster New World College Dictionary, John Wiley and Sons, 4th Edition, May 2004.
51. F. Mendoza, J.L. Bernal-Agustin, J.A. Dominguez-Navarro, "NSGA and SPEA Applied to Multi-Objective Design of Power Distribution Systems", *IEEE Transactions on Power Systems*, Volume 21, No. 4, November 2003 Page(s): 1938-1945.
52. R.T.F.A. King, H.C.S. Rughooputh, K. Deb, "Stochastic Evolutionary Multi-Objective Environmental/Economic Dispatch", *IEEE Congress on Evolutionary Computation* , 16-21 July 2006 Page(s): 946-995.
53. R.T.F.A. King, H.C.S. Rughooputh, "Environmental/Economic Dispatch of Thermal Units using an Elitist Multi-Objective Evolutionary Algorithm.", *IEEE International Conference on Industrial Technology*, Volume 1, No. 1, 10-12 Dec. 2003 Page(s): 48-53.
54. B. Milosevic, M. Begovic, "Capacitor Placement for Conservative Voltage Reduction on Distribution Feeders", *IEEE Transactions on Power Delivery*, Volume 19, No. 3, July 2004 Page(s): 1360-1367.
55. M.A. Abido**,** "A New Multi-Objective Evolutionary Algorithm for Environmental/Economic Power Dispatch", *IEEE Power Engineering Society Summer Meeting*, 2001, Volume 2, 15-19 July 2001 Page(s): 1263-1268.
56. M.A. Abido, "Multi-Objective Evolutionary Algorithms for Electric Power Dispatch", *IEEE Transactions on Evolutionary Computation*, Volume 10, No. 3, June 2006 Page(s): 315-329.

Appendix A – Source Code

MOTS Algorithm

```
/* Memory allocation and deallocation routines */
# include <stdio.h>
# include <stdlib.h>
# include <math.h>
# include "global.h"
# include "rand.h"
/* Function to allocate memory to variables of an individual */
void allocate (individual *ind)
{
   int j;
   if (nbits != 0)
   {
      ind->xreal = (double **)malloc(nbits*sizeof(double));
      ind->gen_cost = (double **)malloc(nbits*sizeof(double));
      ind->gene = (int *)malloc(nbits*sizeof(int));
      for (hr=0; hr<24; j++)
      {
       ind->xreal[hr] = (double *)malloc(nbits*sizeof(double));
       ind->gen_cost[hr] = (double *)malloc(nbits*sizeof(double))
      }
   }

   ind->obj = (double *)malloc(nobj*sizeof(double));
   ind->ia = (int *)malloc(nobj*sizeof(int));
   if (ncon != 0)
   {
      ind->constr = (double *)malloc(ncon*sizeof(double));
   }
   right_genes = (double *)malloc((nbits-site1)*sizeof(double));
   max_gen = (double *)malloc((nbits-site1)*sizeof(double));
   assinged = (double *)malloc(nbits*sizeof(double));
   return;
}

/* Function to deallocate memory of variables of an individual */
void deallocate (individual *ind)
{
   int j;
   if (nbits != 0)
   {
      for (hr=0; hr<24; j++)
      {
```

```
        free(ind->xreal[hr]);
        free(ind->gen_cost[hr]);
        }
        free(ind->xreal);
        free(ind->gen_cost);

        free(ind->gene);


    }
    free(ind->obj);
    free(ind->ia);
    if (ncon != 0)
    {
        free(ind->constr);
    }
    free(right_genes);
    free(max_gen);
    free(assigned);
    return;
}


/* Routine for mergeing two populations */

# include <stdio.h>
# include <stdlib.h>
# include <math.h>

# include "global.h"
# include "rand.h"

/* Routine to copy an individual 'ind1' into another individual 'ind2' */
void copy (individual *ind1, individual *ind2)
{
    int i, j;
    ind2->constr_violation = ind1->constr_violation;
    if (nreal!=0)
    {
        for (i=0; i<nreal; i++)
        {
            ind2->xreal[i] = ind1->xreal[i];
        }
    }
    if (nbits!=0)
    {
```

```
            for (j=0; j<nbits; j++)
            {
               ind2->gene[j] = ind1->gene[j];
               for ( hr =0; hr <24; hr++)
                  ind2->xreal[j][hr] = ind1->xreal[j][hr];
                  ind2->gen_cost[j][hr] = ind1->gen_cost[j];
            }

      }
      for (i=0; i<nobj; i++)
      {
         ind2->obj[i] = ind1->obj[i];
         ind2->ia[i] = ind1->ia[i];
      }
      if (ncon!=0)
      {
         for (i=0; i<ncon; i++)
         {
            ind2->constr[i] = ind1->constr[i];
         }
      }
      return;
}
/* Crossover routines */

# include <stdio.h>
# include <stdlib.h>
# include <math.h>

# include "global.h"
# include "rand.h"

/* Function to cross two individuals */
void crossover (individual *parent1, individual *parent2, individual *child1, individual *child2)
{

   if (nbits!=0)
   {
      bincross (parent1, parent2, child1, child2);
   }
   return;
}


/* Routine for single point binary crossover */
```

```
void bincross (individual *parent1, individual *parent2, individual *child1, individual *child2)
{
   int i, j;
   double rand;
   int temp, site1;

      rand = randomperc();
      if (rand <= pcross_bin)
      {
         nbincross++;

         for (j=0; j<site1; j++)
         {
            child1->gene[j] = parent1->gene[j];
            child2->gene[j] = parent2->gene[j];
            for ( hr= 0 ; hr < 24; hr++)
                {
                child1->xreal[j][hr]= parent1->xreal[j][hr];
                child2->xreal[j][hr]= parent2->xreal[j][hr];
                }
         }

         for (j=site1; j<nbits; j++)
         {
            child1->gene[j] = parent2->gene[j];
            child2->gene[j] = parent1->gene[j];
            for ( hr= 0 ; hr < 24; hr++)
                {
                child1->xreal[j][hr]= parent2->xreal[j][hr];
                child2->xreal[j][hr]= parent1->xreal[j][hr];
                }
         }
      }
      else
      {
         for (j=0; j<nbits; j++)
         {
            child1->gene[j] = parent1->gene[j];
            child2->gene[j] = parent2->gene[j];
            for ( hr= 0 ; hr < 24; hr++)
                {
                child1->xreal[j][hr]= parent1->xreal[j][hr];
                child2->xreal[j][hr]= parent2->xreal[j][hr];
                }
         }
      }
```

```
   return;
}

/* Domination checking routines */

# include <stdio.h>
# include <stdlib.h>
# include <math.h>

# include "global.h"
# include "rand.h"

/*      It returns the following
        1 if a dominates b
        2 if b dominates a
        3 if a and b are non-dominated and a!=b (identification arrays unequal)
        4 if a and b are non-dominated and a=b */

int check_box_dominance (individual *a, individual *b)
{
   int i;
   int flag1;
   int flag2;
   flag1 = 0;
   flag2 = 0;
   if (a->constr_violation<0.0 && b->constr_violation<0.0)
   {
      if (a->constr_violation > b->constr_violation)
      {
         return (1);
      }
      else
      {
         if (a->constr_violation < b->constr_violation)
         {
            return (2);
         }
         else
         {
            return (4);
         }
      }
   }
   else
   {
```

```
if (a->constr_violation<0.0 && b->constr_violation==0.0)
{
   return (2);
}
else
{
   if (a->constr_violation==0.0 && b->constr_violation<0.0)
   {
      return (1);
   }
   else
   {
      for (i=0; i<nobj; i++)
      {
         if (a->ia[i] < b->ia[i])
         {
            flag1 = 1;

         }
         else
         {
            if (a->ia[i] > b->ia[i])
            {
               flag2 = 1;
            }
         }
      }
      if (flag1==1 && flag2==0)
      {
         return (1);
      }
      else
      {
         if (flag1==0 && flag2==1)
         {
            return (2);
         }
         else
         {
            if (flag1==1 && flag2==1)
            {
               return(3);
            }
            else
            {
               return(4);
```

```
                }
            }
        }
    }
  }
}

/* Routine for usual non-domination checking
   It will return the following values
   1 if a dominates b
   -1 if b dominates a
   0 if both a and b are non-dominated */

int check_dominance (individual *a, individual *b)
{
   int i;
   int flag1;
   int flag2;
   flag1 = 0;
   flag2 = 0;
   if (a->constr_violation<0.0 && b->constr_violation<0.0)
   {
      if (a->constr_violation > b->constr_violation)
      {
         return (1);
      }
      else
      {
         if (a->constr_violation < b->constr_violation)
         {
            return (-1);
         }
         else
         {
            return (0);
         }
      }
   }
   else
   {
      if (a->constr_violation<0.0 && b->constr_violation==0.0)
      {
         return (-1);
      }
      else
```

```
        {
          if (a->constr_violation==0.0 && b->constr_violation<0.0)
          {
             return (1);
          }
          else
          {
             for (i=0; i<nobj; i++)
             {
                if (a->obj[i] < b->obj[i])
                {
                   flag1 = 1;

                }
                else
                {
                   if (a->obj[i] > b->obj[i])
                   {
                      flag2 = 1;
                   }
                }
             }
             if (flag1==1 && flag2==0)
             {
                return (1);
             }
             else
             {
                if (flag1==0 && flag2==1)
                {
                   return (-1);
                }
                else
                {
                   return (0);
                }
             }
          }
        }
    }
}

/* EPS-MOTS routine (implementation of the 'main' function) */
/*
 *
 */
```

```
# include <stdio.h>
# include <stdlib.h>
# include <math.h>

# include "global.h"
# include "rand.h"

int nreal;
int nbin;
int nobj;
int ncon;
int popsize;
double pcross_real;
double pcross_bin;
double pmut_real;
double pmut_bin;
double eta_c;
double eta_m;
int neval;
int currenteval;
int nbinmut;
int nrealmut;
int nbincross;
int nrealcross;
int *nbits;
int *array;
double *min_realvar;
double *max_realvar;
double *min_binvar;
double *max_binvar;
double *epsilon;
double *min_obj;
int bitlength;
int elite_size;

int main (int argc, char **argv)
{
   int i;
   int index, index1, index2;
   FILE *fpt1;
   FILE *fpt2;
   FILE *fpt3;
   FILE *fpt4;
   FILE *fpt5;
   individual *ea;
   individual *parent1, *parent2, *child1, *child2;
```

```
ind_list *elite, *cur;
if (argc<2)
{
   printf("\n Usage ./main random_seed \n");
   exit(1);
}
seed = (double)atof(argv[1]);
if (seed<=0.0 || seed>=1.0)
{
   printf("\n Entered seed value is wrong, seed value must be in (0,1) \n");
   exit(1);
}
fpt1 = fopen("initial_pop.out","w");
fpt2 = fopen("final_pop.out","w");
fpt3 = fopen("final_archive.out","w");
fpt4 = fopen("all_archive.out","w");
fpt5 = fopen("params.out","w");
fprintf(fpt1,"# This file contains the data of initial population\n");
fprintf(fpt2,"# This file contains the data of final population\n");
fprintf(fpt3,"# This file contains the best obtained solution(s)\n");
fprintf(fpt4,"# This file contains the data of archive for all generations\n");
fprintf(fpt5,"# This file contains information about inputs as read by the program\n");
printf("\n Enter the problem relevant and algorithm relevant parameters ... ");
printf("\n Enter the population size (>1) : ");
scanf("%d",&popsize);
if (popsize<2)
{
   printf("\n population size read is : %d",popsize);
   printf("\n Wrong population size entered, hence exiting \n");
   exit (1);
}
printf("\n Enter the number of function evaluations : ");
scanf("%d",&neval);
if (neval<popsize)
{
   printf("\n number of function evaluations read is : %d",neval);
   printf("\n Wrong nuber of evaluations entered, hence exiting \n");
   exit (1);
}
printf("\n Enter the number of objectives (>=2): ");
scanf("%d",&nobj);
if (nobj<2)
{
   printf("\n number of objectives entered is : %d",nobj);
   printf("\n Wrong number of objectives entered, hence exiting \n");
   exit (1);
```

```
}
epsilon = (double *)malloc(nobj*sizeof(double));
min_obj = (double *)malloc(nobj*sizeof(double));
for (i=0; i<nobj; i++)
{
   printf("\n Enter the value of epsilon[%d] : ",i+1);
   scanf("%lf",&epsilon[i]);
   if (epsilon[i]<=0.0)
   {
      printf("\n Entered value of epsilon[%d] is non-positive, hence exiting\n",i+1);
      exit(1);
   }
   printf("\n Enter the value of min_obj[%d] (if not known, enter 0.0) : ",i+1);
   scanf("%lf",&min_obj[i]);
}
printf("\n Enter the number of constraints : ");
scanf("%d",&ncon);
if (ncon<0)
{
   printf("\n number of constraints entered is : %d",ncon);
   printf("\n Wrong number of constraints enetered, hence exiting \n");
   exit (1);
}
printf("\n Enter the number of generators : ");
scanf("%d",&nbits);
if (nbits<0)
{
   printf("\n number of real generators entered is : %d",nbits);
   printf("\n Wrong number of generators entered, hence exiting \n");
   exit (1);
}
if (nbits != 0)
{
   min_realvar = (double *)malloc(nreal*sizeof(double));
   max_realvar = (double *)malloc(nreal*sizeof(double));
   for (i=0; i<nbits; i++)
   {
      for ( hr=0; hr <24; hr++)
         {
         printf ("\n Enter the output for generator %d : ",i+1);
         scanf ("%lf",&xreal[i][hr]);
         printf ("\n Enter the cost rate for generator %d : ",i+1);
         scanf ("%lf",&gen_cost[i][hr]);
         }
      printf ("\n Enter the lower limit of real variable %d : ",i+1);
      scanf ("%lf",&min_realvar[i]);
```

```
            printf ("\n Enter the upper limit of real variable %d : ",i+1);
            scanf ("%lf",&max_realvar[i]);
            if (max_realvar[i] <= min_realvar[i])
            {
               printf("\n Wrong limits entered for the min and max bounds of generator %d, hence
exiting \n",i+1);
               exit(1);
            }
         }
         printf ("\n Enter the probability of crossover (0.6-1.0) : ");
         scanf ("%lf",&pcross_real);
         if (pcross_real<0.0 || pcross_real>1.0)
         {
            printf("\n Probability of crossover entered is : %e",pcross_real);
            printf("\n Entered value of probability of crossover of real variables is out of bounds,
hence exiting \n");
            exit (1);
         }
         printf ("\n Enter the probablity of mutation (1/nreal) : ");
         scanf ("%lf",&pmut_real);
         if (pmut_real<0.0 || pmut_real>1.0)
         {
            printf("\n Probability of mutation entered is : %e",pmut_real);
            printf("\n Entered value of probability of mutation of real variables is out of bounds,
hence exiting \n");
            exit (1);
         }
         printf ("\n Enter the value of distribution index for crossover (5-20): ");
         scanf ("%lf",&eta_c);
         if (eta_c<=0)
         {
            printf("\n The value entered is : %e",eta_c);
            printf("\n Wrong value of distribution index for crossover entered, hence exiting \n");
            exit (1);
         }
         printf ("\n Enter the value of distribution index for mutation (5-50): ");
         scanf ("%lf",&eta_m);
         if (eta_m<=0)
         {
            printf("\n The value entered is : %e",eta_m);
            printf("\n Wrong value of distribution index for mutation entered, hence exiting \n");
            exit (1);
         }
      }
```

```
   if (nbits==0)
   {
      printf("\n Number of variables is zero, hence exiting \n");
      exit(1);
   }
   printf("\n Input data successfully entered, now performing initialization \n");
   fprintf(fpt5,"\n Population size = %d",popsize);
   fprintf(fpt5,"\n Number of function evaluations = %d",neval);
   fprintf(fpt5,"\n Number of objective functions = %d",nobj);
   for (i=0; i<nobj; i++)
   {
      fprintf(fpt5,"\n Epsilon for objective %d = %e",i+1,epsilon[i]);
      fprintf(fpt5,"\n Minimum value of objective %d = %e",i+1,min_obj[i]);
   }
   fprintf(fpt5,"\n Number of constraints = %d",ncon);
   fprintf(fpt5,"\n Number of real variables = %d",nreal);
   if (nreal!=0)
   {
      for (i=0; i<nbits; i++)
      {
         fprintf(fpt5,"\n Lower limit of real variable %d = %e",i+1,min_realvar[i]);
         fprintf(fpt5,"\n Upper limit of real variable %d = %e",i+1,max_realvar[i]);
      }
      fprintf(fpt5,"\n Probability of crossover of real variable = %e",pcross_real);
      fprintf(fpt5,"\n Probability of mutation of real variable = %e",pmut_real);
      fprintf(fpt5,"\n Distribution index for crossover = %e",eta_c);
      fprintf(fpt5,"\n Distribution index for mutation = %e",eta_m);
   }
   fprintf(fpt5,"\n Number of binary variables = %d",nbin);

   fprintf(fpt5,"\n Seed for random number generator = %e",seed);
   bitlength = 0;

   fprintf(fpt1,"# of objectives = %d, # of constraints = %d, # of real_var = %d, # of bits of
bin_var = %d, constr_violation\n",nobj,ncon,nreal,bitlength);
   fprintf(fpt2,"# of objectives = %d, # of constraints = %d, # of real_var = %d, # of bits of
bin_var = %d, constr_violation\n",nobj,ncon,nreal,bitlength);
   fprintf(fpt3,"# of objectives = %d, # of constraints = %d, # of real_var = %d, # of bits of
bin_var = %d, constr_violation\n",nobj,ncon,nreal,bitlength);
   fprintf(fpt4,"# of objectives = %d, # of constraints = %d, # of real_var = %d, # of bits of
bin_var = %d, constr_violation\n",nobj,ncon,nreal,bitlength);
   nbinmut = 0;
   nrealmut = 0;
   nbincross = 0;
   nrealcross = 0;
   currenteval = 0;
```

```
elite_size = 0;
randomize();
ea = (individual *)malloc(popsize*sizeof(individual));
array = (int *)malloc(popsize*sizeof(int));
for (i=0; i<popsize; i++)
{
   allocate (&ea[i]);
   initialize(&ea[i]);
   decode(&ea[i]);
   eval(&ea[i]);
}
report_pop (ea, fpt1);
elite = (ind_list *)malloc(sizeof(ind_list));
elite->ind = (individual *)malloc(sizeof(individual));
allocate (elite->ind);
elite->parent = NULL;
elite->child = NULL;
insert (elite, &ea[0]);
for (i=1; i<popsize; i++)
{
   update_elite (elite, &ea[i]);
}
child1 = (individual *)malloc(sizeof(individual));
allocate (child1);
child2 = (individual *)malloc(sizeof(individual));
allocate (child2);
cur = elite;
while (currenteval<neval)
{
   index1 = rnd(0, popsize-1);
   index2 = rnd(0, popsize-1);
   parent1 = tournament (&ea[index1], &ea[index2]);
   index = rnd(0, elite_size-1);
   cur = elite->child;
   for (i=1; i<=index; i++)
   {
      cur=cur->child;
   }
   parent2 = cur->ind;
   crossover (parent1, parent2, child1, child2);
   mutation (child1);
   decode (child1);
   eval (child1);
   update_elite (elite, child1);
   update_pop (ea, child1);
   mutation (child2);
```

```
    decode (child2);
    eval (child2);
    update_elite (elite, child2);
    update_pop (ea, child2);
    printf("\n Currenteval = %d and Elite_size = %d",currenteval,elite_size);
                /* Comment following three lines if information at all
                evaluation is not desired, it will speed up execution of the code */
       fprintf(fpt4,"# eval id = %d\n",currenteval);
                report_archive (elite, fpt4);
                fflush(fpt4);
}
printf("\n Generations finished, now reporting solutions");
report_pop (ea, fpt2);
report_archive (elite, fpt3);
if (nreal!=0)
{
    fprintf(fpt5,"\n Number of crossover of real variable = %d",nrealcross);
    fprintf(fpt5,"\n Number of mutation of real variable = %d",nrealmut);
}
if (nbin!=0)
{
    fprintf(fpt5,"\n Number of crossover of binary variable = %d",nbincross);
    fprintf(fpt5,"\n Number of mutation of binary variable = %d",nbinmut);
}
fflush(stdout);
fflush(fpt1);
fflush(fpt2);
fflush(fpt3);
fflush(fpt4);
fflush(fpt5);
fclose(fpt1);
fclose(fpt2);
fclose(fpt3);
fclose(fpt4);
fclose(fpt5);
if (nreal!=0)
{
    free (min_realvar);
    free (max_realvar);
}
if (nbin!=0)
{
    free (min_binvar);
    free (max_binvar);
    free (nbits);
}
```

```
    free (epsilon);
    free (min_obj);
    free (array);
    for (i=0; i<popsize; i++)
    {
        deallocate (&ea[i]);
    }
    free (ea);
    cur = elite->child;
    while (cur!=NULL)
    {
        cur = del(cur);
        cur = cur->child;
    }
    deallocate (elite->ind);
    free (elite->ind);
    free (elite);
    printf("\n Routine successfully exited \n");
    return (0);
}

/* Routine for evaluating individuals  */

# include <stdio.h>
# include <stdlib.h>
# include <math.h>

# include "global.h"
# include "rand.h"

/* Routine to evaluate objective function values and constraints for an individual */
void eval (individual *ind)
{
    int j;
    test_problem (ind->xreal,ind->gen_cost, ind->gene, ind->obj, ind->constr);
    for (j=0; j<nobj; j++)
    {
        ind->ia[j] = (int)floor( (ind->obj[j]-min_obj[j])/epsilon[j] );
    }
    if (ncon==0)
    {
        ind->constr_violation = 0.0;
    }
    else
    {
        ind->constr_violation = 0.0;
```

```
        for (j=0; j<ncon; j++)
        {
            if (ind->constr[j]<0.0)
            {
                ind->constr_violation += ind->constr[j];
            }
        }
    }
    currenteval++;
    return;
}

/* This file contains the variable and function declarations */

# ifndef _GLOBAL_H_
# define _GLOBAL_H_

# define INF 1.0e99
# define EPS 1.0e-14
# define E  2.71828182845905
# define PI 3.14159265358979

/* global variables */
typedef struct
{
    double constr_violation;
    double **xreal;
    int *gene;
    double *gen_cost;
    double *obj;
    int *ia;
    double *constr;
} individual;

typedef struct ind_lists
{
    individual *ind;
    struct ind_lists *parent;
    struct ind_lists *child;
} ind_list;

extern int nreal;
extern int nbin;
extern int nobj;
extern int ncon;
extern int popsize;
```

```
extern double pcross_real;
extern double pcross_bin;
extern double pmut_real;
extern double pmut_bin;
extern double eta_c;
extern double eta_m;
extern int neval;
extern int currenteval;
extern int nbinmut;
extern int nrealmut;
extern int nbincross;
extern int nrealcross;
extern int nbits;
extern int *array;
extern int *right_genes;
extern int *assigned;
extern double max_gen;
extern double *min_realvar;
extern double *max_realvar;
extern double *min_binvar;
extern double *max_binvar;
extern double *epsilon;
extern double *min_obj;
extern int bitlength;
extern int elite_size;
extern int site1;
/* global function declarations */
void allocate (individual *ind);
void deallocate (individual *ind);

void copy (individual *ind1, individual *ind2);

void crossover (individual *parent1, individual *parent2, individual *child1, individual *child2);
void realcross (individual *parent1, individual *parent2, individual *child1, individual *child2);
void bincross (individual *parent1, individual *parent2, individual *child1, individual *child2);

void decode (individual *ind);

int check_box_dominance (individual *a, individual *b);
int check_dominance (individual *a, individual *b);

void eval (individual *ind);

void initialize (individual *ind);

void insert (ind_list *node, individual *ind);
```

```
ind_list* del (ind_list *node);

void mutation (individual *ind);
void bin_mutate (individual *ind);
void real_mutate (individual *ind);

void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr);

void report_pop (individual *ind, FILE *fpt);
void report_archive (ind_list *elite, FILE *fpt);

individual* tournament (individual *ind1, individual *ind2);

void update_elite (ind_list *elite, individual *ind);
void update_pop (individual *ea, individual *ind);

# endif

/* Data initializtion routines */

# include <stdio.h>
# include <stdlib.h>
# include <math.h>

# include "global.h"
# include "rand.h"

/* Function to initialize an individual randomly */
void initialize (individual *ind, int site1)
{
   int i, j, k, found, found2,res;
   if (nbits !=0)
   {

   for (i=0; i < (nbits - site1); i++)
      {
      max_gen[i]= 0.0;
      for (j=0; j<nbits; j++)
         {
         if (i == 0)
            {
             if (max_gen[i] < gen_cost[j])
                max_gen[i]= gen_cost[j];
            }
         else
            {
```

```
          if ((max_gen[i] < gen_cost[j]) && (max_gen[i] < max_gen[i-1]))
            max_gen[i]= gen_cost[j];
          }
        }
    /*fetch a single random integer at a time to select each gene to the right of the crossover point
and is one of the expensive generators */
    k =0;
    j= site1;
    right_genes[0] =10000;

        while (k < nbits -site1)
            {
            res = rnd[0,nbits];

               if ( gen_cost[res] >= max_gen[nbits -site1-1])
                  {
                  found =0;
                   for (i=0; i < k ; i++)
                      if (res == right_genes[i])
                        found=1;
                      if ( found == 0)
                        {
                        ind->gene[j] = res;
                        for ( hr=0; hr <24; hr++)
                           ind->xreal[j][hr]= rndreal (min_realvar[res], max_realvar[res]);
                        right_genes[k] = res;
                        j++;
                        k++;
                        }
                  }
           }
      /* next assign all remaining generators not assigned to the appropreate gene */
      jump=k;
      j=0;
      assigned[0]=10000;
      while ( j < site1)
         {
         res = rnd[0,bits];
         found =0;
         found2=0;
         for (k=0 ; k < jump; k++)
            if( res == right_genes[k])
              found =1;

         for ( i=0; i < j; i++)
            if( res == assigned[i] )
```

```
                found2=1;
                   if ( found1 ==0 && found2 ==0)
                     {
                     ind->gene[j] = res;
                     for ( hr= 0 ; hr < 24; hr++)
                        {
                        ind->xreal[j][hr]= rndreal (min_realvar[res], max_realvar[res]);

                        }
                     assigned [j] =res;
                     j++;
                     }


          }
   }
   return;
}

/* A custom doubly linked list implemenation */

# include <stdio.h>
# include <stdlib.h>
# include <math.h>

# include "global.h"
# include "rand.h"

/* Routine to insert an element after the location specified by node NODE */
void insert (ind_list *node, individual *ind)
{
   ind_list *temp;
   if (node==NULL)
   {
      printf("\n Error!! asked to enter after a NULL pointer, hence exiting \n");
      exit(1);
   }
   temp = (ind_list *)malloc(sizeof(ind_list));
   temp->ind = (individual *)malloc(sizeof(individual));
   allocate (temp->ind);
   copy (ind, temp->ind);
   temp->child = node->child;
   temp->parent = node;
   if (node->child != NULL)
   {
      node->child->parent = temp;
```

```
    }
    node->child = temp;
    elite_size++;
    return;
}

/* Delete the element specified by node NODE */
ind_list* del (ind_list *node)
{
    ind_list *temp;
    if (node==NULL)
    {
        printf("\n Error!! asked to delete a NULL pointer, hence exiting \n");
        exit(1);
    }
    temp = node->parent;
    temp->child = node->child;
    if (temp->child!=NULL)
    {
        temp->child->parent = temp;
    }
    deallocate(node->ind);
    free (node->ind);
    free (node);
    elite_size--;
    return (temp);
}

/* Mutation routines */

# include <stdio.h>
# include <stdlib.h>
# include <math.h>

# include "global.h"
# include "rand.h"

/* Function to perform mutation of an individual */
void mutation (individual *ind)
{
    if (nreal!=0)
    {
        real_mutate(ind);
    }
    if (nbits!=0)
    {
```

```
      bin_mutate(ind);
   }
   return;
}

/* Routine for binary mutation of an individual */
void bin_mutate (individual *ind)
{
   int j, k, found,found2,temp;
   double prob, temp_real[24],new_real[24];


   for( hr=0; hr <24; hr++)
     {
     temp_real[24]=0.0;
     new_real[24]=0.0;
     }

        prob = randomperc();
        if (prob <=pmut_bin)
        {
        found =0;
        res = rnd[0,site1];
        while (found !=1 )
           {
           res1 = rnd[0,site1];
           if ( res !=res1)
             {

             for( hr=0; hr <24; hr++)
               temp_real[24] = ind->xreal[res][hr]+ind->xreal[res1][hr];
             temp = ind->gene[res];
             ind-> gene[res] = ind->gene[res1];
             ind-> gene[res1] = temp;
             for( hr=0; hr <24; hr++)
               {
               found2=0;
               while( found2 !=1)
                   {
                   ind->xreal[res][hr] =rndreal (min_realvar[res], max_realvar[res]);
                   new_real[24] = temp_real[24] - ind->xreal[res][hr];

                   if ( new_real[24] > min_realvar[res1])&&( new_real[24] < max_realvar[res1]))
                     {
                     ind->xreal[res1][hr] = new_real[24];
                     found2=1;
```

```
                    }
                  }
                }
              found=1;
              }
            }
          }

   return;
}

/* Routine for real polynomial mutation of an individual */
void real_mutate (individual *ind)
{
   int j;
   double rnd, delta1, delta2, mut_pow, deltaq;
   double y, yl, yu, val, xy;
   for (j=0; j<nreal; j++)
   {
                if (randomperc() <= pmut_real)
      {
         y = ind->xreal[j];
         yl = min_realvar[j];
         yu = max_realvar[j];
         delta1 = (y-yl)/(yu-yl);
         delta2 = (yu-y)/(yu-yl);
         rnd = randomperc();
         mut_pow = 1.0/(eta_m+1.0);
         if (rnd <= 0.5)
         {
            xy = 1.0-delta1;
            val = 2.0*rnd+(1.0-2.0*rnd)*(pow(xy,(eta_m+1.0)));
            deltaq =  pow(val,mut_pow) - 1.0;
         }
         else
         {
            xy = 1.0-delta2;
            val = 2.0*(1.0-rnd)+2.0*(rnd-0.5)*(pow(xy,(eta_m+1.0)));
            deltaq = 1.0 - (pow(val,mut_pow));
         }
         y = y + deltaq*(yu-yl);
         if (y<yl)          y = yl;
                            if (y>yu)          y = yu;
                            ind->xreal[j] = y;
                            nrealmut+=1;
                }
```

```
    }
        return;
}

/* Test problem definitions */

# include <stdio.h>
# include <stdlib.h>
# include <math.h>

# include "global.h"
# include "rand.h"

/* # define sch1 */
/* # define sch2 */
/* # define fon */
/* # define kur */
/* # define pol */
/* # define vnt */
/* # define zdt1*/
/* # define zdt2 */
/* # define zdt3 */
/* # define zdt4 */
/* # define zdt5 */
/* # define zdt6 */
/* # define bnh */
/* # define osy */
/* # define srn */
/* # define tnk */
/* # define ctp1 */
/* # define ctp2 */
/* # define ctp3 */
/* # define ctp4 */
/* # define ctp5 */
/* # define ctp6 */
/* # define ctp7 */
/* # define ctp8 */
#define generator_matchup
/*  Test problem SCH1
    # of real variables = 1
    # of bin variables = 0
    # of objectives = 2
    # of constraints = 0
    */

#ifdef sch1
```

```
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   obj[0] = pow(xreal[0],2.0);
   obj[1] = pow((xreal[0]-2.0),2.0);
   return;
}
#endif

/*  Test problem SCH2
   # of real variables = 1
   # of bin variables = 0
   # of objectives = 2
   # of constraints = 0
   */

#ifdef sch2
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   if (xreal[0]<=1.0)
   {
      obj[0] = -xreal[0];
      obj[1] = pow((xreal[0]-5.0),2.0);
      return;
   }
   if (xreal[0]<=3.0)
   {
      obj[0] = xreal[0]-2.0;
      obj[1] = pow((xreal[0]-5.0),2.0);
      return;
   }
   if (xreal[0]<=4.0)
   {
      obj[0] = 4.0-xreal[0];
      obj[1] = pow((xreal[0]-5.0),2.0);
      return;
   }
   obj[0] = xreal[0]-4.0;
   obj[1] = pow((xreal[0]-5.0),2.0);
   return;
}
#endif

/*  Test problem FON
   # of real variables = n
   # of bin variables = 0
   # of objectives = 2
```

```
   # of constraints = 0
   */

#ifdef fon
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   double s1, s2;
   int i;
   s1 = s2 = 0.0;
   for (i=0; i<nreal; i++)
   {
      s1 += pow((xreal[i]-(1.0/sqrt((double)nreal))),2.0);
      s2 += pow((xreal[i]+(1.0/sqrt((double)nreal))),2.0);
   }
   obj[0] = 1.0 - exp(-s1);
   obj[1] = 1.0 - exp(-s2);
   return;
}
#endif

/*  Test problem KUR
   # of real variables = 3
   # of bin variables = 0
   # of objectives = 2
   # of constraints = 0
   */

#ifdef kur
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   int i;
   double res1, res2;
   res1 = -0.2*sqrt((xreal[0]*xreal[0]) + (xreal[1]*xreal[1]));
   res2 = -0.2*sqrt((xreal[1]*xreal[1]) + (xreal[2]*xreal[2]));
   obj[0] = -10.0*( exp(res1) + exp(res2));
   obj[1] = 0.0;
   for (i=0; i<3; i++)
   {
      obj[1] += pow(fabs(xreal[i]),0.8) + 5.0*sin(pow(xreal[i],3.0));
   }
   return;
}
#endif

/*  Test problem POL
   # of real variables = 2
```

```
   # of bin variables = 0
   # of objectives = 2
   # of constraints = 0
   */

#ifdef pol
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   double a1, a2, b1, b2;
   a1 = 0.5*sin(1.0) - 2.0*cos(1.0) + sin(2.0) - 1.5*cos(2.0);
   a2 = 1.5*sin(1.0) - cos(1.0) + 2.0*sin(2.0) - 0.5*cos(2.0);
   b1 = 0.5*sin(xreal[0]) - 2.0*cos(xreal[0]) + sin(xreal[1]) - 1.5*cos(xreal[1]);
   b2 = 1.5*sin(xreal[0]) - cos(xreal[0]) + 2.0*sin(xreal[1]) - 0.5*cos(xreal[1]);
   obj[0] = 1.0 + pow((a1-b1),2.0) + pow((a2-b2),2.0);
   obj[1] = pow((xreal[0]+3.0),2.0) + pow((xreal[1]+1.0),2.0);
   return;
}
#endif

/*  Test problem VNT
   # of real variables = 2
   # of bin variables = 0
   # of objectives = 3
   # of constraints = 0
   */

#ifdef vnt
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   obj[0] = 0.5*(xreal[0]*xreal[0] + xreal[1]*xreal[1]) + sin(xreal[0]*xreal[0] +
xreal[1]*xreal[1]);
   obj[1] = (pow((3.0*xreal[0] - 2.0*xreal[1] + 4.0),2.0))/8.0 + (pow((xreal[0]-
xreal[1]+1.0),2.0))/27.0 + 15.0;
   obj[2] = 1.0/(xreal[0]*xreal[0] + xreal[1]*xreal[1] + 1.0) - 1.1*exp(-(xreal[0]*xreal[0] +
xreal[1]*xreal[1]));
   return;
}
#endif

/*  Test problem ZDT1
   # of real variables = 30
   # of bin variables = 0
   # of objectives = 2
   # of constraints = 0
   */
```

```
#ifdef zdt1
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   double f1, f2, g, h;
   int i;
   f1 = xreal[0];
   g = 0.0;
   for (i=1; i<30; i++)
   {
      g += xreal[i];
   }
   g = 9.0*g/29.0;
   g += 1.0;
   h = 1.0 - sqrt(f1/g);
   f2 = g*h;
   obj[0] = f1;
   obj[1] = f2;
   return;
}
#endif

/*  Test problem ZDT2
   # of real variables = 30
   # of bin variables = 0
   # of objectives = 2
   # of constraints = 0
   */

#ifdef zdt2
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   double f1, f2, g, h;
   int i;
   f1 = xreal[0];
   g = 0.0;
   for (i=1; i<30; i++)
   {
      g += xreal[i];
   }
   g = 9.0*g/29.0;
   g += 1.0;
   h = 1.0 - pow((f1/g),2.0);
   f2 = g*h;
   obj[0] = f1;
   obj[1] = f2;
   return;
```

```
}
#endif

/*  Test problem ZDT3
   # of real variables = 30
   # of bin variables = 0
   # of objectives = 2
   # of constraints = 0
   */

#ifdef zdt3
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   double f1, f2, g, h;
   int i;
   f1 = xreal[0];
   g = 0.0;
   for (i=1; i<30; i++)
   {
      g += xreal[i];
   }
   g = 9.0*g/29.0;
   g += 1.0;
   h = 1.0 - sqrt(f1/g) - (f1/g)*sin(10.0*PI*f1);
   f2 = g*h;
   obj[0] = f1;
   obj[1] = f2;
   return;
}
#endif

/*  Test problem ZDT4
   # of real variables = 10
   # of bin variables = 0
   # of objectives = 2
   # of constraints = 0
   */

#ifdef zdt4
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   double f1, f2, g, h;
   int i;
   f1 = xreal[0];
   g = 0.0;
   for (i=1; i<10; i++)
```

```
    {
        g += xreal[i]*xreal[i] - 10.0*cos(4.0*PI*xreal[i]);
    }
    g += 91.0;
    h = 1.0 - sqrt(f1/g);
    f2 = g*h;
    obj[0] = f1;
    obj[1] = f2;
    return;
}
#endif

/*  Test problem ZDT5
    # of real variables = 0
    # of bin variables = 11
    # of bits for binvar1 = 30
    # of bits for binvar2-11 = 5
    # of objectives = 2
    # of constraints = 0
    */

#ifdef zdt5
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
    int i, j;
    int u[11];
    int v[11];
    double f1, f2, g, h;
    for (i=0; i<11; i++)
    {
        u[i] = 0;
    }
    for (j=0; j<30; j++)
    {
        if (gene[0][j] == 1)
        {
            u[0]++;
        }
    }
    for (i=1; i<11; i++)
    {
        for (j=0; j<4; j++)
        {
            if (gene[i][j] == 1)
            {
                u[i]++;
```

```
            }
        }
    }
    f1 = 1.0 + u[0];
    for (i=1; i<11; i++)
    {
        if (u[i] < 5)
        {
            v[i] = 2 + u[i];
        }
        else
        {
            v[i] = 1;
        }
    }
    g = 0;
    for (i=1; i<11; i++)
    {
        g += v[i];
    }
    h = 1.0/f1;
    f2 = g*h;
    obj[0] = f1;
    obj[1] = f2;
    return;
}
#endif

/*  Test problem ZDT6
    # of real variables = 10
    # of bin variables = 0
    # of objectives = 2
    # of constraints = 0
    */

#ifdef zdt6
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
    double f1, f2, g, h;
    int i;
    f1 = 1.0 - (exp(-4.0*xreal[0]))*pow((sin(4.0*PI*xreal[0])),6.0);
    g = 0.0;
    for (i=1; i<10; i++)
    {
        g += xreal[i];
    }
```

```
    g = g/9.0;
    g = pow(g,0.25);
    g = 1.0 + 9.0*g;
    h = 1.0 - pow((f1/g),2.0);
    f2 = g*h;
    obj[0] = f1;
    obj[1] = f2;
    return;
}
#endif

/*  Test problem BNH
    # of real variables = 2
    # of bin variables = 0
    # of objectives = 2
    # of constraints = 2
    */

#ifdef bnh
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
    obj[0] = 4.0*(xreal[0]*xreal[0] + xreal[1]*xreal[1]);
    obj[1] = pow((xreal[0]-5.0),2.0) + pow((xreal[1]-5.0),2.0);
    constr[0] = 1.0 - (pow((xreal[0]-5.0),2.0) + xreal[1]*xreal[1])/25.0;
    constr[1] = (pow((xreal[0]-8.0),2.0) + pow((xreal[1]+3.0),2.0))/7.7 - 1.0;
    return;
}
#endif

/*  Test problem OSY
    # of real variables = 6
    # of bin variables = 0
    # of objectives = 2
    # of constraints = 6
    */

#ifdef osy
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
    obj[0] = -(25.0*pow((xreal[0]-2.0),2.0) + pow((xreal[1]-2.0),2.0) + pow((xreal[2]-1.0),2.0) +
pow((xreal[3]-4.0),2.0) + pow((xreal[4]-1.0),2.0));
    obj[1] = xreal[0]*xreal[0] +  xreal[1]*xreal[1] + xreal[2]*xreal[2] + xreal[3]*xreal[3] +
xreal[4]*xreal[4] + xreal[5]*xreal[5];
    constr[0] = (xreal[0]+xreal[1])/2.0 - 1.0;
    constr[1] = 1.0 - (xreal[0]+xreal[1])/6.0;
    constr[2] = 1.0 - xreal[1]/2.0 + xreal[0]/2.0;
```

```
   constr[3] = 1.0 - xreal[0]/2.0 + 3.0*xreal[1]/2.0;
   constr[4] = 1.0 - (pow((xreal[2]-3.0),2.0))/4.0 - xreal[3]/4.0;
   constr[5] = (pow((xreal[4]-3.0),2.0))/4.0 + xreal[5]/4.0 - 1.0;
   return;
}
#endif

/*  Test problem SRN
   # of real variables = 2
   # of bin variables = 0
   # of objectives = 2
   # of constraints = 2
   */

#ifdef srn
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   obj[0] = 2.0 + pow((xreal[0]-2.0),2.0) + pow((xreal[1]-1.0),2.0);
   obj[1] = 9.0*xreal[0] - pow((xreal[1]-1.0),2.0);
   constr[0] = 1.0 - (pow(xreal[0],2.0) + pow(xreal[1],2.0))/225.0;
   constr[1] = 3.0*xreal[1]/10.0 - xreal[0]/10.0 - 1.0;
   return;
}
#endif

/*  Test problem TNK
   # of real variables = 2
   # of bin variables = 0
   # of objectives = 2
   # of constraints = 2
   */

#ifdef tnk
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   obj[0] = xreal[0];
   obj[1] = xreal[1];
   if (xreal[1] == 0.0)
   {
      constr[0] = -1.0;
   }
   else
   {
      constr[0] = xreal[0]*xreal[0] + xreal[1]*xreal[1] - 0.1*cos(16.0*atan(xreal[0]/xreal[1])) -
1.0;
   }
```

```
   constr[1] = 1.0 - 2.0*pow((xreal[0]-0.5),2.0) + 2.0*pow((xreal[1]-0.5),2.0);
   return;
}
#endif

/*  Test problem CTP1
   # of real variables = 2
   # of bin variables = 0
   # of objectives = 2
   # of constraints = 2
   */

#ifdef ctp1
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   double g;
   g = 1.0 + xreal[1];
   obj[0] = xreal[0];
   obj[1] = g*exp(-obj[0]/g);
   constr[0] = obj[1]/(0.858*exp(-0.541*obj[0]))-1.0;
   constr[1] = obj[1]/(0.728*exp(-0.295*obj[0]))-1.0;
   return;
}
#endif

/*  Test problem CTP2
   # of real variables = 2
   # of bin variables = 0
   # of objectives = 2
   # of constraints = 1
   */

#ifdef ctp2
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   double g;
   double theta, a, b, c, d, e;
   double exp1, exp2;
   theta = -0.2*PI;
   a = 0.2;
   b = 10.0;
   c = 1.0;
   d = 6.0;
   e = 1.0;
   g = 1.0 + xreal[1];
   obj[0] = xreal[0];
```

```
    obj[1] = g*(1.0  - sqrt(obj[0]/g));
    exp1 = (obj[1]-e)*cos(theta) - obj[0]*sin(theta);
    exp2 = (obj[1]-e)*sin(theta) + obj[0]*cos(theta);
    exp2 = b*PI*pow(exp2,c);
    exp2 = fabs(sin(exp2));
    exp2 = a*pow(exp2,d);
    constr[0] = exp1/exp2 - 1.0;
    return;
}
#endif

/*  Test problem CTP3
    # of real variables = 2
    # of bin variables = 0
    # of objectives = 2
    # of constraints = 1
    */

#ifdef ctp3
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
    double g;
    double theta, a, b, c, d, e;
    double exp1, exp2;
    theta = -0.2*PI;
    a = 0.1;
    b = 10.0;
    c = 1.0;
    d = 0.5;
    e = 1.0;
    g = 1.0 + xreal[1];
    obj[0] = xreal[0];
    obj[1] = g*(1.0  - sqrt(obj[0]/g));
    exp1 = (obj[1]-e)*cos(theta) - obj[0]*sin(theta);
    exp2 = (obj[1]-e)*sin(theta) + obj[0]*cos(theta);
    exp2 = b*PI*pow(exp2,c);
    exp2 = fabs(sin(exp2));
    exp2 = a*pow(exp2,d);
    constr[0] = exp1/exp2 - 1.0;
    return;
}
#endif

/*  Test problem CTP4
    # of real variables = 2
    # of bin variables = 0
```

```
   # of objectives = 2
   # of constraints = 1
   */

#ifdef ctp4
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   double g;
   double theta, a, b, c, d, e;
   double exp1, exp2;
   theta = -0.2*PI;
   a = 0.75;
   b = 10.0;
   c = 1.0;
   d = 0.5;
   e = 1.0;
   g = 1.0 + xreal[1];
   obj[0] = xreal[0];
   obj[1] = g*(1.0  - sqrt(obj[0]/g));
   exp1 = (obj[1]-e)*cos(theta) - obj[0]*sin(theta);
   exp2 = (obj[1]-e)*sin(theta) + obj[0]*cos(theta);
   exp2 = b*PI*pow(exp2,c);
   exp2 = fabs(sin(exp2));
   exp2 = a*pow(exp2,d);
   constr[0] = exp1/exp2 - 1.0;
   return;
}
#endif

/*  Test problem CTP5
   # of real variables = 2
   # of bin variables = 0
   # of objectives = 2
   # of constraints = 1
   */

#ifdef ctp5
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   double g;
   double theta, a, b, c, d, e;
   double exp1, exp2;
   theta = -0.2*PI;
   a = 0.1;
   b = 10.0;
   c = 2.0;
```

```
    d = 0.5;
    e = 1.0;
    g = 1.0 + xreal[1];
    obj[0] = xreal[0];
    obj[1] = g*(1.0  - sqrt(obj[0]/g));
    exp1 = (obj[1]-e)*cos(theta) - obj[0]*sin(theta);
    exp2 = (obj[1]-e)*sin(theta) + obj[0]*cos(theta);
    exp2 = b*PI*pow(exp2,c);
    exp2 = fabs(sin(exp2));
    exp2 = a*pow(exp2,d);
    constr[0] = exp1/exp2 - 1.0;
    return;
}
#endif

/*  Test problem CTP6
    # of real variables = 2
    # of bin variables = 0
    # of objectives = 2
    # of constraints = 1
    */

#ifdef ctp6
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
    double g;
    double theta, a, b, c, d, e;
    double exp1, exp2;
    theta = 0.1*PI;
    a = 40.0;
    b = 0.5;
    c = 1.0;
    d = 2.0;
    e = -2.0;
    g = 1.0 + xreal[1];
    obj[0] = xreal[0];
    obj[1] = g*(1.0  - sqrt(obj[0]/g));
    exp1 = (obj[1]-e)*cos(theta) - obj[0]*sin(theta);
    exp2 = (obj[1]-e)*sin(theta) + obj[0]*cos(theta);
    exp2 = b*PI*pow(exp2,c);
    exp2 = fabs(sin(exp2));
    exp2 = a*pow(exp2,d);
    constr[0] = exp1/exp2 - 1.0;
    return;
}
#endif
```

```
/*  Test problem CTP7
    # of real variables = 2
    # of bin variables = 0
    # of objectives = 2
    # of constraints = 1
    */

#ifdef ctp7
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
    double g;
    double theta, a, b, c, d, e;
    double exp1, exp2;
    theta = -0.05*PI;
    a = 40.0;
    b = 5.0;
    c = 1.0;
    d = 6.0;
    e = 0.0;
    g = 1.0 + xreal[1];
    obj[0] = xreal[0];
    obj[1] = g*(1.0  - sqrt(obj[0]/g));
    exp1 = (obj[1]-e)*cos(theta) - obj[0]*sin(theta);
    exp2 = (obj[1]-e)*sin(theta) + obj[0]*cos(theta);
    exp2 = b*PI*pow(exp2,c);
    exp2 = fabs(sin(exp2));
    exp2 = a*pow(exp2,d);
    constr[0] = exp1/exp2 - 1.0;
    return;
}
#endif

/*  Test problem CTP8
    # of real variables = 2
    # of bin variables = 0
    # of objectives = 2
    # of constraints = 2
    */

#ifdef ctp8
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
    double g;
    double theta, a, b, c, d, e;
    double exp1, exp2;
```

```
    g = 1.0 + xreal[1];
    obj[0] = xreal[0];
    obj[1] = g*(1.0  - sqrt(obj[0]/g));
    theta = 0.1*PI;
    a = 40.0;
    b = 0.5;
    c = 1.0;
    d = 2.0;
    e = -2.0;
    exp1 = (obj[1]-e)*cos(theta) - obj[0]*sin(theta);
    exp2 = (obj[1]-e)*sin(theta) + obj[0]*cos(theta);
    exp2 = b*PI*pow(exp2,c);
    exp2 = fabs(sin(exp2));
    exp2 = a*pow(exp2,d);
    constr[0] = exp1/exp2 - 1.0;
    theta = -0.05*PI;
    a = 40.0;
    b = 2.0;
    c = 1.0;
    d = 6.0;
    e = 0.0;
    exp1 = (obj[1]-e)*cos(theta) - obj[0]*sin(theta);
    exp2 = (obj[1]-e)*sin(theta) + obj[0]*cos(theta);
    exp2 = b*PI*pow(exp2,c);
    exp2 = fabs(sin(exp2));
    exp2 = a*pow(exp2,d);
    constr[1] = exp1/exp2 - 1.0;
    return;
}
#endif

#ifdef generator_matchup
void test_problem (double **xreal,double *gen_cost, double *load_cost, double **load, double
*max_realvar, int num_loads, int nreal, int *group, int group_max, double *obj, double *constr)
{
  int i,j,k,l, hrs=24;
  obj[0] =0.0;
  obj[1] = 0.0;
double sum =0.0, sum1=0.0;
/* Objective function definition */

    for (i = 0; i < hrs ; i++)
        {
        for ( j = 0; j < nreal ; j++)
            {
            obj[0] += (gen_cost[j]*xreal[j][i];
```

```
            }
        for ( k= 0; k < num_loads ; k++)
            {
            obj[0] -= load_cost[k]*load[k][i];
            }
        }
    for (i = 0; i < hrs ; i++)
        {
        for ( j = 0; j < nreal ; j++)
            {
            for( l = 0; l < group_max ; l++)
              {
              if ( j == group[l])
                sum1 += max_realvar[j] - xreal[j][i];
              }
            sum += max_realvar[j] -x real[j][i];
            }
         for ( j = 0; j < nreal ; j++)
            {
            for( l = 0; l < group_max ; l++)
              {
              if ( j != group[l])
                 obj[1] += ((max_realvar[j]- xreal[j][i])/sum)^2;
              }
            }
          obj[1] += (sum1/sum)^2;
        }
       obj[1] = obj[1]/24;
/* constraints definition */
/* Economic Minimum and Maximum Operating Cosntaraint *

    for (i = 0; i < hrs ; i++)
        {
        for ( j = 0; j < nreal ; j++)
            {
            constr[j]=max_realvar[j] -(xreal[j][i]+ Spin[j];
            }
        }
    for (i = 0; i < hrs ; i++)
        {
        for ( j = 0; j < nreal ; j++)
            {
            constr[j+nreal]=max_realvar[j] -(xreal[j][i]+ SOper[j];
            }
        }
    for (i = 0; i < hrs ; i++)
```

```
        {
        for ( j = 0; j < nreal ; j++)
            {
            constr[j+2*nreal]=(xreal[j][i] -min_realvar[j];
            }
        }


    return;
}
#endif

/* Definition of random number generation routines */

# include <stdio.h>
# include <stdlib.h>
# include <math.h>

# include "global.h"
# include "rand.h"

double seed;
double oldrand[55];
int jrand;

/* Get seed number for random and start it up */
void randomize()
{
    int j1;
    for(j1=0; j1<=54; j1++)
    {
        oldrand[j1] = 0.0;
    }
    jrand=0;
    warmup_random (seed);
    return;
}

/* Get randomize off and running */
void warmup_random (double seed)
{
    int j1, ii;
    double new_random, prev_random;
    oldrand[54] = seed;
    new_random = 0.000000001;
    prev_random = seed;
```

```
   for(j1=1; j1<=54; j1++)
   {
      ii = (21*j1)%54;
      oldrand[ii] = new_random;
      new_random = prev_random-new_random;
      if(new_random<0.0)
      {
         new_random += 1.0;
      }
      prev_random = oldrand[ii];
   }
   advance_random ();
   advance_random ();
   advance_random ();
   jrand = 0;
   return;
}

/* Create next batch of 55 random numbers */
void advance_random ()
{
   int j1;
   double new_random;
   for(j1=0; j1<24; j1++)
   {
      new_random = oldrand[j1]-oldrand[j1+31];
      if(new_random<0.0)
      {
         new_random = new_random+1.0;
      }
      oldrand[j1] = new_random;
   }
   for(j1=24; j1<55; j1++)
   {
      new_random = oldrand[j1]-oldrand[j1-24];
      if(new_random<0.0)
      {
         new_random = new_random+1.0;
      }
      oldrand[j1] = new_random;
   }
}

/* Fetch a single random number between 0.0 and 1.0 */
double randomperc()
{
```

```
   jrand++;
   if(jrand>=55)
   {
      jrand = 1;
      advance_random();
   }
   return((double)oldrand[jrand]);
}

/* Fetch a single random integer between low and high including the bounds */
int rnd (int low, int high)
{
   int res;
   if (low >= high)
   {
      res = low;
   }
   else
   {
      res = low + (randomperc()*(high-low+1));
      if (res > high)
      {
         res = high;
      }
   }
   return (res);
}

/* Fetch a single random real number between low and high including the bounds */
double rndreal (double low, double high)
{
   return (low + (high-low)*randomperc());
}

/* Declaration for random number related variables and routines */

# ifndef _RAND_H_
# define _RAND_H_

/* Variable declarations for the random number generator */
extern double seed;
extern double oldrand[55];
extern int jrand;

/* Function declarations for the random number generator */
void randomize(void);
```

```
void warmup_random (double seed);
void advance_random (void);
double randomperc(void);
int rnd (int low, int high);
double rndreal (double low, double high);

# endif

/* Routines for storing population data into files */

# include <stdio.h>
# include <stdlib.h>
# include <math.h>

# include "global.h"
# include "rand.h"

/* Function to print the information of a population in a file */
void report_pop (individual *ind, FILE *fpt)
{
   int i, j, k;
   for (i=0; i<popsize; i++)
   {
      for (j=0; j<nobj; j++)
      {
         fprintf(fpt,"%e\t",ind[i].obj[j]);
      }
      if (ncon!=0)
      {
         for (j=0; j<ncon; j++)
         {
            fprintf(fpt,"%e\t",ind[i].constr[j]);
         }
      }
      if (nreal!=0)
      {
         for (j=0; j<nbits; j++)
         {
            for ( hr <0; hr <24; hr++)
            fprintf(fpt,"%e\t",cur->ind->xreal[j][hr]);
            fprintf(fpt,"\n");
         }
      }
      if (nbits!=0)
      {
```

```
            for (k=0; k<nbits; k++)
            {
               fprintf(fpt,"%d\t",ind[i].gene[k]);
            }

         }
         fprintf(fpt,"%e\n",ind[i].constr_violation);
      }
      return;
}

/* Function to print the information of feasible and non-dominated population in a file */
void report_archive (ind_list *elite, FILE *fpt)
{
      int j, k;
      ind_list *cur;
      cur = elite->child;
      while (cur!=NULL)
      {
         for (j=0; j<nobj; j++)
         {
            fprintf(fpt,"%e\t",cur->ind->obj[j]);
         }
         if (ncon!=0)
         {
            for (j=0; j<ncon; j++)
            {
               fprintf(fpt,"%e\t",cur->ind->constr[j]);
            }
         }
         if (nbits!=0)
         {
            for (j=0; j<nbits; j++)
            {
             for ( hr <0; hr <24; hr++)
               fprintf(fpt,"%e\t",cur->ind->xreal[j][hr]);
               fprintf(fpt,"\n");
            }
         }
         if (nbits!=0)
         {

               for (k=0; k<nbits; k++)
               {
                  fprintf(fpt,"%d\t",cur->ind->gene[k]);
               }
```

```
        }
        fprintf(fpt,"%e\n",cur->ind->constr_violation);
        cur = cur->child;
    }
    return;
}

/* Tournamenet Selections routine */

# include <stdio.h>
# include <stdlib.h>
# include <math.h>

# include "global.h"
# include "rand.h"

/* Routine for binary neigborhood */
individual* tournament (individual *ind1, individual *ind2)
{
    int flag;
    flag = check_dominance (ind1, ind2);
    if (flag==1)
    {
        return (ind1);
    }
    if (flag==-1)
    {
        return (ind2);
    }
    if ((randomperc()) <= 0.5)
    {
        return(ind1);
    }
    else
    {
        return(ind2);
    }
}

/* Routines for updating elite and EA populations */

# include <stdio.h>
# include <stdlib.h>
# include <math.h>
```

```
# include "global.h"
# include "rand.h"

/* Routine to update archive */
void update_elite (ind_list *elite, individual *ind)
{
   int i, end, flag;
   double d1, d2;
   ind_list *temp;
   temp = elite->child;
   end = 0;
   do
   {
      flag = check_box_dominance (ind, temp->ind);
      switch (flag)
      {
      case 1: /* ind dominates temp->ind */
         {
            temp = del (temp);
            temp = temp->child;
            break;
         }
      case 2: /* temp->ind dominates ind */
         {
            return;
         }
      case 3: /* both are non-dominated and are in different boxes */
         {
            temp = temp->child;
            break;
         }
      case 4: /* both are non-dominated and are in same hyper-box */
         {
            end = 1;
            break;
         }
      }
   }
   while (end!=1 && temp!=NULL);
   if (end==0)
   {
      insert(elite, ind);
   }
   else
   {
      if (flag==4)       /* in same hyperbox */
```

```
        {
            flag = check_dominance (ind, temp->ind);
            switch (flag)
            {
            case 1:
                {
                    temp = del(temp);
                    insert (elite, ind);
                    break;
                }
            case -1:
                {
                    return;
                }
            case 0:
                {
                    d1 = 0.0;
                    d2 = 0.0;
                    for (i=0; i<nobj; i++)
                    {
                        d1 += pow(((ind->obj[i]-ind->ia[i])/epsilon[i]),2.0);
                        d2 += pow(((temp->ind->obj[i]-temp->ind->ia[i])/epsilon[i]),2.0);
                    }
                    if (d1<=d2)
                    {
                        temp = del(temp);
                        insert(elite,ind);
                    }
                    break;
                }
            }
        }
    }
    return;
}

/* Routine to update population */
void update_pop (individual *ea, individual *ind)
{
        int size;
        int i;
        int flag;
        size = 0;
        for (i=0; i<popsize; i++)
        {
                flag = check_dominance (ind, &ea[i]);
```

```
                switch (flag)
                {
                        case 1:
                                copy (ind, &ea[i]);
                                return;
                        case -1:
                                return;
                        case 0:
                                array[size++] = i;
                                break;
                }
        }
        if (size>0)
        {
                i = rnd(0,size-1);
                copy (ind, &ea[array[i]]);
        }
        return;
}
```

MOTS/GA Algorithm

```
/* Memory allocation and deallocation routines */
# include <stdio.h>
# include <stdlib.h>
# include <math.h>
# include "global.h"
# include "rand.h"
/* Function to allocate memory to variables of an individual */
void allocate (individual *ind)
{
   int j;
   if (nbits != 0)
   {
      ind->xreal = (double **)malloc(nbits*sizeof(double));
      ind->gen_cost = (double **)malloc(nbits*sizeof(double));
      ind->gene = (int *)malloc(nbits*sizeof(int));
      for (hr=0; hr<24; j++)
      {
        ind->xreal[hr] = (double *)malloc(nbits*sizeof(double));
        ind->gen_cost[hr] = (double *)malloc(nbits*sizeof(double))
      }
   }
```

```
   ind->obj = (double *)malloc(nobj*sizeof(double));
   ind->ia = (int *)malloc(nobj*sizeof(int));
   if (ncon != 0)
   {
      ind->constr = (double *)malloc(ncon*sizeof(double));
   }
   right_genes = (double *)malloc((nbits-site1)*sizeof(double));
   max_gen = (double *)malloc((nbits-site1)*sizeof(double));
   assinged = (double *)malloc(nbits*sizeof(double));
   return;
}

/* Function to deallocate memory of variables of an individual */
void deallocate (individual *ind)
{
   int j;
   if (nbits != 0)
   {
      for (hr=0; hr<24; j++)
      {
       free(ind->xreal[hr]);
       free(ind->gen_cost[hr]);
      }
      free(ind->xreal);
      free(ind->gen_cost);

      free(ind->gene);


   }
   free(ind->obj);
   free(ind->ia);
   if (ncon != 0)
   {
      free(ind->constr);
   }
   free(right_genes);
   free(max_gen);
   free(assigned);
   return;
}


/* Routine for mergeing two populations */

# include <stdio.h>
```

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

# include "global.h"
# include "rand.h"

/* Routine to copy an individual 'ind1' into another individual 'ind2' */
void copy (individual *ind1, individual *ind2)
{
   int i, j;
   ind2->constr_violation = ind1->constr_violation;
   if (nreal!=0)
   {
      for (i=0; i<nreal; i++)
      {
         ind2->xreal[i] = ind1->xreal[i];
      }
   }
   if (nbits!=0)
   {

         for (j=0; j<nbits; j++)
         {
            ind2->gene[j] = ind1->gene[j];
            for ( hr =0; hr <24; hr++)
               ind2->xreal[j][hr] = ind1->xreal[j][hr];
               ind2->gen_cost[j][hr] = ind1->gen_cost[j];
         }

   }
   for (i=0; i<nobj; i++)
   {
      ind2->obj[i] = ind1->obj[i];
      ind2->ia[i] = ind1->ia[i];
   }
   if (ncon!=0)
   {
      for (i=0; i<ncon; i++)
      {
         ind2->constr[i] = ind1->constr[i];
      }
   }
   return;
}
/* Crossover routines */
```

```
# include <stdio.h>
# include <stdlib.h>
# include <math.h>

# include "global.h"
# include "rand.h"

/* Function to cross two individuals */
void crossover (individual *parent1, individual *parent2, individual *child1, individual *child2)
{

   if (nbits!=0)
   {
      bincross (parent1, parent2, child1, child2);
   }
   return;
}


/* Routine for single point binary crossover */
void bincross (individual *parent1, individual *parent2, individual *child1, individual *child2)
{
   int i, j;
   double rand;
   int temp, site1;

      rand = randomperc();
      if (rand <= pcross_bin)
      {
         nbincross++;

         for (j=0; j<site1; j++)
         {
            child1->gene[j] = parent1->gene[j];
            child2->gene[j] = parent2->gene[j];
            for ( hr= 0 ; hr < 24; hr++)
                {
                child1->xreal[j][hr]= parent1->xreal[j][hr];
                child2->xreal[j][hr]= parent2->xreal[j][hr];
                }
         }

         for (j=site1; j<nbits; j++)
         {
            child1->gene[j] = parent2->gene[j];
            child2->gene[j] = parent1->gene[j];
```

```
            for ( hr= 0 ; hr < 24; hr++)
                {
                child1->xreal[j][hr]= parent2->xreal[j][hr];
                child2->xreal[j][hr]= parent1->xreal[j][hr];
                }
            }
        }
        else
        {
            for (j=0; j<nbits; j++)
            {
                child1->gene[j] = parent1->gene[j];
                child2->gene[j] = parent2->gene[j];
                for ( hr= 0 ; hr < 24; hr++)
                    {
                    child1->xreal[j][hr]= parent1->xreal[j][hr];
                    child2->xreal[j][hr]= parent2->xreal[j][hr];
                    }
            }
        }

    return;
}

/* Domination checking routines */

# include <stdio.h>
# include <stdlib.h>
# include <math.h>

# include "global.h"
# include "rand.h"

/*      It returns the following
        1 if a dominates b
        2 if b dominates a
        3 if a and b are non-dominated and a!=b (identification arrays unequal)
        4 if a and b are non-dominated and a=b */

int check_box_dominance (individual *a, individual *b)
{
    int i;
    int flag1;
    int flag2;
    flag1 = 0;
    flag2 = 0;
```

```
if (a->constr_violation<0.0 && b->constr_violation<0.0)
{
   if (a->constr_violation > b->constr_violation)
   {
      return (1);
   }
   else
   {
      if (a->constr_violation < b->constr_violation)
      {
         return (2);
      }
      else
      {
         return (4);
      }
   }
}
else
{
   if (a->constr_violation<0.0 && b->constr_violation==0.0)
   {
      return (2);
   }
   else
   {
      if (a->constr_violation==0.0 && b->constr_violation<0.0)
      {
         return (1);
      }
      else
      {
         for (i=0; i<nobj; i++)
         {
            if (a->ia[i] < b->ia[i])
            {
               flag1 = 1;

            }
            else
            {
               if (a->ia[i] > b->ia[i])
               {
                  flag2 = 1;
               }
            }
```

```
            }
            if (flag1==1 && flag2==0)
            {
               return (1);
            }
            else
            {
               if (flag1==0 && flag2==1)
               {
                  return (2);
               }
               else
               {
                  if (flag1==1 && flag2==1)
                  {
                     return(3);
                  }
                  else
                  {
                     return(4);
                  }
               }
            }
         }
      }
   }
}

/* Routine for usual non-domination checking
   It will return the following values
   1 if a dominates b
   -1 if b dominates a
   0 if both a and b are non-dominated */

int check_dominance (individual *a, individual *b)
{
   int i;
   int flag1;
   int flag2;
   flag1 = 0;
   flag2 = 0;
   if (a->constr_violation<0.0 && b->constr_violation<0.0)
   {
      if (a->constr_violation > b->constr_violation)
      {
         return (1);
```

```
    }
    else
    {
        if (a->constr_violation < b->constr_violation)
        {
            return (-1);
        }
        else
        {
            return (0);
        }
    }
}
else
{
    if (a->constr_violation<0.0 && b->constr_violation==0.0)
    {
        return (-1);
    }
    else
    {
        if (a->constr_violation==0.0 && b->constr_violation<0.0)
        {
            return (1);
        }
        else
        {
            for (i=0; i<nobj; i++)
            {
                if (a->obj[i] < b->obj[i])
                {
                    flag1 = 1;

                }
                else
                {
                    if (a->obj[i] > b->obj[i])
                    {
                        flag2 = 1;
                    }
                }
            }
            if (flag1==1 && flag2==0)
            {
                return (1);
            }
```

```
            else
            {
               if (flag1==0 && flag2==1)
               {
                  return (-1);
               }
               else
               {
                  return (0);
               }
            }
         }
      }
   }
}

/* EPS-MOTS routine (implementation of the 'main' function) */
/*
 *
 */
# include <stdio.h>
# include <stdlib.h>
# include <math.h>

# include "global.h"
# include "rand.h"

int nreal;
int nbin;
int nobj;
int ncon;
int popsize;
double pcross_real;
double pcross_bin;
double pmut_real;
double pmut_bin;
double eta_c;
double eta_m;
int neval;
int currenteval;
int nbinmut;
int nrealmut;
int nbincross;
int nrealcross;
int *nbits;
int *array;
```

```
double *min_realvar;
double *max_realvar;
double *min_binvar;
double *max_binvar;
double *epsilon;
double *min_obj;
int bitlength;
int elite_size;

int main (int argc, char **argv)
{
   int i;
   int index, index1, index2;
   FILE *fpt1;
   FILE *fpt2;
   FILE *fpt3;
   FILE *fpt4;
   FILE *fpt5;
   individual *ea;
   individual *parent1, *parent2, *child1, *child2;
   ind_list *elite, *cur;
   if (argc<2)
   {
      printf("\n Usage ./main random_seed \n");
      exit(1);
   }
   seed = (double)atof(argv[1]);
   if (seed<=0.0 || seed>=1.0)
   {
      printf("\n Entered seed value is wrong, seed value must be in (0,1) \n");
      exit(1);
   }
   fpt1 = fopen("initial_pop.out","w");
   fpt2 = fopen("final_pop.out","w");
   fpt3 = fopen("final_archive.out","w");
   fpt4 = fopen("all_archive.out","w");
   fpt5 = fopen("params.out","w");
   fprintf(fpt1,"# This file contains the data of initial population\n");
   fprintf(fpt2,"# This file contains the data of final population\n");
   fprintf(fpt3,"# This file contains the best obtained solution(s)\n");
   fprintf(fpt4,"# This file contains the data of archive for all generations\n");
   fprintf(fpt5,"# This file contains information about inputs as read by the program\n");
   printf("\n Enter the problem relevant and algorithm relevant parameters ... ");
   printf("\n Enter the population size (>1) : ");
   scanf("%d",&popsize);
   if (popsize<2)
```

```
{
   printf("\n population size read is : %d",popsize);
   printf("\n Wrong population size entered, hence exiting \n");
   exit (1);
}
printf("\n Enter the number of function evaluations : ");
scanf("%d",&neval);
if (neval<popsize)
{
   printf("\n number of function evaluations read is : %d",neval);
   printf("\n Wrong nuber of evaluations entered, hence exiting \n");
   exit (1);
}
printf("\n Enter the number of objectives (>=2): ");
scanf("%d",&nobj);
if (nobj<2)
{
   printf("\n number of objectives entered is : %d",nobj);
   printf("\n Wrong number of objectives entered, hence exiting \n");
   exit (1);
}
epsilon = (double *)malloc(nobj*sizeof(double));
min_obj = (double *)malloc(nobj*sizeof(double));
for (i=0; i<nobj; i++)
{
   printf("\n Enter the value of epsilon[%d] : ",i+1);
   scanf("%lf",&epsilon[i]);
   if (epsilon[i]<=0.0)
   {
      printf("\n Entered value of epsilon[%d] is non-positive, hence exiting\n",i+1);
      exit(1);
   }
   printf("\n Enter the value of min_obj[%d] (if not known, enter 0.0) : ",i+1);
   scanf("%lf",&min_obj[i]);
}
printf("\n Enter the number of constraints : ");
scanf("%d",&ncon);
if (ncon<0)
{
   printf("\n number of constraints entered is : %d",ncon);
   printf("\n Wrong number of constraints enetered, hence exiting \n");
   exit (1);
}
printf("\n Enter the number of generators : ");
scanf("%d",&nbits);
if (nbits<0)
```

```
    {
       printf("\n number of real generators entered is : %d",nbits);
       printf("\n Wrong number of generators entered, hence exiting \n");
       exit (1);
    }
    if (nbits != 0)
    {
       min_realvar = (double *)malloc(nreal*sizeof(double));
       max_realvar = (double *)malloc(nreal*sizeof(double));
       for (i=0; i<nbits; i++)
       {
          for ( hr=0; hr <24; hr++)
             {
             printf ("\n Enter the output for generator %d : ",i+1);
             scanf ("%lf",&xreal[i][hr]);
             printf ("\n Enter the cost rate for generator %d : ",i+1);
             scanf ("%lf",&gen_cost[i][hr]);
             }
          printf ("\n Enter the lower limit of real variable %d : ",i+1);
          scanf ("%lf",&min_realvar[i]);
          printf ("\n Enter the upper limit of real variable %d : ",i+1);
          scanf ("%lf",&max_realvar[i]);
          if (max_realvar[i] <= min_realvar[i])
          {
             printf("\n Wrong limits entered for the min and max bounds of generator %d, hence
exiting \n",i+1);
             exit(1);
          }
       }
       printf ("\n Enter the probability of crossover (0.6-1.0) : ");
       scanf ("%lf",&pcross_real);
       if (pcross_real<0.0 || pcross_real>1.0)
       {
          printf("\n Probability of crossover entered is : %e",pcross_real);
          printf("\n Entered value of probability of crossover of real variables is out of bounds,
hence exiting \n");
          exit (1);
       }
       printf ("\n Enter the probablity of mutation (1/nreal) : ");
       scanf ("%lf",&pmut_real);
       if (pmut_real<0.0 || pmut_real>1.0)
       {
          printf("\n Probability of mutation entered is : %e",pmut_real);
          printf("\n Entered value of probability of mutation of real variables is out of bounds,
hence exiting \n");
          exit (1);
```

```
    }
    printf ("\n Enter the value of distribution index for crossover (5-20): ");
    scanf ("%lf",&eta_c);
    if (eta_c<=0)
    {
        printf("\n The value entered is : %e",eta_c);
        printf("\n Wrong value of distribution index for crossover entered, hence exiting \n");
        exit (1);
    }
    printf ("\n Enter the value of distribution index for mutation (5-50): ");
    scanf ("%lf",&eta_m);
    if (eta_m<=0)
    {
        printf("\n The value entered is : %e",eta_m);
        printf("\n Wrong value of distribution index for mutation entered, hence exiting \n");
        exit (1);
    }
}


if (nbits==0)
{
    printf("\n Number of variables is zero, hence exiting \n");
    exit(1);
}
printf("\n Input data successfully entered, now performing initialization \n");
fprintf(fpt5,"\n Population size = %d",popsize);
fprintf(fpt5,"\n Number of function evaluations = %d",neval);
fprintf(fpt5,"\n Number of objective functions = %d",nobj);
for (i=0; i<nobj; i++)
{
    fprintf(fpt5,"\n Epsilon for objective %d = %e",i+1,epsilon[i]);
    fprintf(fpt5,"\n Minimum value of objective %d = %e",i+1,min_obj[i]);
}
fprintf(fpt5,"\n Number of constraints = %d",ncon);
fprintf(fpt5,"\n Number of real variables = %d",nreal);
if (nreal!=0)
{
    for (i=0; i<nbits; i++)
    {
        fprintf(fpt5,"\n Lower limit of real variable %d = %e",i+1,min_realvar[i]);
        fprintf(fpt5,"\n Upper limit of real variable %d = %e",i+1,max_realvar[i]);
    }
    fprintf(fpt5,"\n Probability of crossover of real variable = %e",pcross_real);
    fprintf(fpt5,"\n Probability of mutation of real variable = %e",pmut_real);
    fprintf(fpt5,"\n Distribution index for crossover = %e",eta_c);
```

```
      fprintf(fpt5,"\n Distribution index for mutation = %e",eta_m);
   }
   fprintf(fpt5,"\n Number of binary variables = %d",nbin);

   fprintf(fpt5,"\n Seed for random number generator = %e",seed);
   bitlength = 0;

   fprintf(fpt1,"# of objectives = %d, # of constraints = %d, # of real_var = %d, # of bits of
bin_var = %d, constr_violation\n",nobj,ncon,nreal,bitlength);
   fprintf(fpt2,"# of objectives = %d, # of constraints = %d, # of real_var = %d, # of bits of
bin_var = %d, constr_violation\n",nobj,ncon,nreal,bitlength);
   fprintf(fpt3,"# of objectives = %d, # of constraints = %d, # of real_var = %d, # of bits of
bin_var = %d, constr_violation\n",nobj,ncon,nreal,bitlength);
   fprintf(fpt4,"# of objectives = %d, # of constraints = %d, # of real_var = %d, # of bits of
bin_var = %d, constr_violation\n",nobj,ncon,nreal,bitlength);
   nbinmut = 0;
   nrealmut = 0;
   nbincross = 0;
   nrealcross = 0;
   currenteval = 0;
   elite_size = 0;
   randomize();
   ea = (individual *)malloc(popsize*sizeof(individual));
   array = (int *)malloc(popsize*sizeof(int));
   for (i=0; i<popsize; i++)
   {
      allocate (&ea[i]);
      initialize(&ea[i]);
      decode(&ea[i]);
      eval(&ea[i]);
   }
   report_pop (ea, fpt1);
   elite = (ind_list *)malloc(sizeof(ind_list));
   elite->ind = (individual *)malloc(sizeof(individual));
   allocate (elite->ind);
   elite->parent = NULL;
   elite->child = NULL;
   insert (elite, &ea[0]);
   for (i=1; i<popsize; i++)
   {
      update_elite (elite, &ea[i]);
   }
   child1 = (individual *)malloc(sizeof(individual));
   allocate (child1);
   child2 = (individual *)malloc(sizeof(individual));
   allocate (child2);
```

```
cur = elite;
while (currenteval<neval)
{
   index1 = rnd(0, popsize-1);
   index2 = rnd(0, popsize-1);
   parent1 = tournament (&ea[index1], &ea[index2]);
   index = rnd(0, elite_size-1);
   cur = elite->child;
   for (i=1; i<=index; i++)
   {
      cur=cur->child;
   }
   parent2 = cur->ind;
   crossover (parent1, parent2, child1, child2);
   mutation (child1);
   decode (child1);
   eval (child1);
   update_elite (elite, child1);
   update_pop (ea, child1);
   mutation (child2);
   decode (child2);
   eval (child2);
   update_elite (elite, child2);
   update_pop (ea, child2);
   printf("\n Currenteval = %d and Elite_size = %d",currenteval,elite_size);
                /* Comment following three lines if information at all
                evaluation is not desired, it will speed up execution of the code */
      fprintf(fpt4,"# eval id = %d\n",currenteval);
                report_archive (elite, fpt4);
                fflush(fpt4);
}
printf("\n Generations finished, now reporting solutions");
report_pop (ea, fpt2);
report_archive (elite, fpt3);
if (nreal!=0)
{
   fprintf(fpt5,"\n Number of crossover of real variable = %d",nrealcross);
   fprintf(fpt5,"\n Number of mutation of real variable = %d",nrealmut);
}
if (nbin!=0)
{
   fprintf(fpt5,"\n Number of crossover of binary variable = %d",nbincross);
   fprintf(fpt5,"\n Number of mutation of binary variable = %d",nbinmut);
}
fflush(stdout);
fflush(fpt1);
```

```
    fflush(fpt2);
    fflush(fpt3);
    fflush(fpt4);
    fflush(fpt5);
    fclose(fpt1);
    fclose(fpt2);
    fclose(fpt3);
    fclose(fpt4);
    fclose(fpt5);
    if (nreal!=0)
    {
       free (min_realvar);
       free (max_realvar);
    }
    if (nbin!=0)
    {
       free (min_binvar);
       free (max_binvar);
       free (nbits);
    }
    free (epsilon);
    free (min_obj);
    free (array);
    for (i=0; i<popsize; i++)
    {
       deallocate (&ea[i]);
    }
    free (ea);
    cur = elite->child;
    while (cur!=NULL)
    {
       cur = del(cur);
       cur = cur->child;
    }
    deallocate (elite->ind);
    free (elite->ind);
    free (elite);
    printf("\n Routine successfully exited \n");
    return (0);
}

/* Routine for evaluating individuals  */

# include <stdio.h>
# include <stdlib.h>
# include <math.h>
```

```
# include "global.h"
# include "rand.h"

/* Routine to evaluate objective function values and constraints for an individual */
void eval (individual *ind)
{
   int j;
   test_problem (ind->xreal,ind->gen_cost, ind->gene, ind->obj, ind->constr);
   for (j=0; j<nobj; j++)
   {
      ind->ia[j] = (int)floor( (ind->obj[j]-min_obj[j])/epsilon[j] );
   }
   if (ncon==0)
   {
      ind->constr_violation = 0.0;
   }
   else
   {
      ind->constr_violation = 0.0;
      for (j=0; j<ncon; j++)
      {
         if (ind->constr[j]<0.0)
         {
            ind->constr_violation += ind->constr[j];
         }
      }
   }
   currenteval++;
   return;
}

/* This file contains the variable and function declarations */

# ifndef _GLOBAL_H_
# define _GLOBAL_H_

# define INF 1.0e99
# define EPS 1.0e-14
# define E  2.71828182845905
# define PI 3.14159265358979

/* global variables */
typedef struct
{
   double constr_violation;
```

```
    double **xreal;
    int *gene;
    double *gen_cost;
    double *obj;
    int *ia;
    double *constr;
} individual;

typedef struct ind_lists
{
    individual *ind;
    struct ind_lists *parent;
    struct ind_lists *child;
} ind_list;

extern int nreal;
extern int nbin;
extern int nobj;
extern int ncon;
extern int popsize;
extern double pcross_real;
extern double pcross_bin;
extern double pmut_real;
extern double pmut_bin;
extern double eta_c;
extern double eta_m;
extern int neval;
extern int currenteval;
extern int nbinmut;
extern int nrealmut;
extern int nbincross;
extern int nrealcross;
extern int nbits;
extern int *array;
extern int *right_genes;
extern int *assigned;
extern double max_gen;
extern double *min_realvar;
extern double *max_realvar;
extern double *min_binvar;
extern double *max_binvar;
extern double *epsilon;
extern double *min_obj;
extern int bitlength;
extern int elite_size;
extern int site1;
```

```
/* global function declarations */
void allocate (individual *ind);
void deallocate (individual *ind);

void copy (individual *ind1, individual *ind2);

void crossover (individual *parent1, individual *parent2, individual *child1, individual *child2);
void realcross (individual *parent1, individual *parent2, individual *child1, individual *child2);
void bincross (individual *parent1, individual *parent2, individual *child1, individual *child2);

void decode (individual *ind);

int check_box_dominance (individual *a, individual *b);
int check_dominance (individual *a, individual *b);

void eval (individual *ind);

void initialize (individual *ind);

void insert (ind_list *node, individual *ind);
ind_list* del (ind_list *node);

void mutation (individual *ind);
void bin_mutate (individual *ind);
void real_mutate (individual *ind);

void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr);

void report_pop (individual *ind, FILE *fpt);
void report_archive (ind_list *elite, FILE *fpt);

individual* tournament (individual *ind1, individual *ind2);

void update_elite (ind_list *elite, individual *ind);
void update_pop (individual *ea, individual *ind);

# endif

/* Data initializtion routines */

# include <stdio.h>
# include <stdlib.h>
# include <math.h>

# include "global.h"
# include "rand.h"
```

```
/* Function to initialize an individual randomly */
void initialize (individual *ind, int site1)
{
   int i, j, k, found, found2,res;
   if (nbits !=0)
   {

   for (i=0; i < (nbits - site1); i++)
      {
      max_gen[i]= 0.0;
      for (j=0; j<nbits; j++)
         {
         if (i == 0)
           {
            if (max_gen[i] < gen_cost[j])
               max_gen[i]= gen_cost[j];
           }
         else
           {
           if ((max_gen[i] < gen_cost[j]) && (max_gen[i] < max_gen[i-1]))
             max_gen[i]= gen_cost[j];
           }
         }
    /*fetch a single random integer at a time to select each gene to the right of the crossover point
and is one of the expensive generators */
    k =0;
    j= site1;
    right_genes[0] =10000;

        while (k < nbits -site1)
            {
            res = rnd[0,nbits];

               if ( gen_cost[res] >= max_gen[nbits -site1-1])
                  {
                  found =0;
                   for (i=0; i < k ; i++)
                      if (res == right_genes[i])
                        found=1;
                      if ( found == 0)
                        {
                        ind->gene[j] = res;
                        for ( hr=0; hr <24; hr++)
                            ind->xreal[j][hr]= rndreal (min_realvar[res], max_realvar[res]);
                        right_genes[k] = res;
```

```
                    j++;
                    k++;
                    }
                }
            }
        /* next assign all remaining generators not assigned to the appropreate gene */
        jump=k;
        j=0;
        assigned[0]=10000;
        while ( j < site1)
            {
            res = rnd[0,bits];
            found =0;
            found2=0;
            for (k=0 ; k < jump; k++)
                if( res == right_genes[k])
                  found =1;

            for ( i=0; i < j; i++)
                if( res == assigned[i] )
                  found2=1;
                    if ( found1 ==0 && found2 ==0)
                      {
                      ind->gene[j] = res;
                      for ( hr= 0 ; hr < 24; hr++)
                          {
                          ind->xreal[j][hr]= rndreal (min_realvar[res], max_realvar[res]);

                          }
                      assigned [j] =res;
                      j++;
                      }


            }
    }
    return;
}

/* A custom doubly linked list implemenation */

# include <stdio.h>
# include <stdlib.h>
# include <math.h>

# include "global.h"
```

```
# include "rand.h"

/* Routine to insert an element after the location specified by node NODE */
void insert (ind_list *node, individual *ind)
{
   ind_list *temp;
   if (node==NULL)
   {
      printf("\n Error!! asked to enter after a NULL pointer, hence exiting \n");
      exit(1);
   }
   temp = (ind_list *)malloc(sizeof(ind_list));
   temp->ind = (individual *)malloc(sizeof(individual));
   allocate (temp->ind);
   copy (ind, temp->ind);
   temp->child = node->child;
   temp->parent = node;
   if (node->child != NULL)
   {
      node->child->parent = temp;
   }
   node->child = temp;
   elite_size++;
   return;
}

/* Delete the element specified by node NODE */
ind_list* del (ind_list *node)
{
   ind_list *temp;
   if (node==NULL)
   {
      printf("\n Error!! asked to delete a NULL pointer, hence exiting \n");
      exit(1);
   }
   temp = node->parent;
   temp->child = node->child;
   if (temp->child!=NULL)
   {
      temp->child->parent = temp;
   }
   deallocate(node->ind);
   free (node->ind);
   free (node);
   elite_size--;
   return (temp);
```

```
}

/* Mutation routines */

# include <stdio.h>
# include <stdlib.h>
# include <math.h>

# include "global.h"
# include "rand.h"

/* Function to perform mutation of an individual */
void mutation (individual *ind)
{
   if (nreal!=0)
   {
      real_mutate(ind);
   }
   if (nbits!=0)
   {
      bin_mutate(ind);
   }
   return;
}

/* Routine for binary mutation of an individual */
void bin_mutate (individual *ind)
{
   int j, k, found,found2,temp;
   double prob, temp_real[24],new_real[24];


   for( hr=0; hr <24; hr++)
     {
     temp_real[24]=0.0;
     new_real[24]=0.0;
     }

         prob = randomperc();
         if (prob <=pmut_bin)
         {
         found =0;
         res = rnd[0,site1];
         while (found !=1 )
            {
            res1 = rnd[0,site1];
```

```
            if ( res !=res1)
              {

              for( hr=0; hr <24; hr++)
                temp_real[24] = ind->xreal[res][hr]+ind->xreal[res1][hr];
              temp = ind->gene[res];
              ind-> gene[res] = ind->gene[res1];
              ind-> gene[res1] = temp;
              for( hr=0; hr <24; hr++)
                {
                found2=0;
                while( found2 !=1)
                    {
                    ind->xreal[res][hr] =rndreal (min_realvar[res], max_realvar[res]);
                    new_real[24] = temp_real[24] - ind->xreal[res][hr];

                    if ( new_real[24] > min_realvar[res1])&&( new_real[24] < max_realvar[res1]))
                      {
                      ind->xreal[res1][hr] = new_real[24];
                      found2=1;
                      }
                    }
                }
              found=1;
              }
            }
        }

   return;
}

/* Routine for real polynomial mutation of an individual */
void real_mutate (individual *ind)
{
   int j;
   double rnd, delta1, delta2, mut_pow, deltaq;
   double y, yl, yu, val, xy;
   for (j=0; j<nreal; j++)
   {
                  if (randomperc() <= pmut_real)
      {
         y = ind->xreal[j];
         yl = min_realvar[j];
         yu = max_realvar[j];
         delta1 = (y-yl)/(yu-yl);
         delta2 = (yu-y)/(yu-yl);
```

```
            rnd = randomperc();
            mut_pow = 1.0/(eta_m+1.0);
            if (rnd <= 0.5)
            {
                xy = 1.0-delta1;
                val = 2.0*rnd+(1.0-2.0*rnd)*(pow(xy,(eta_m+1.0)));
                deltaq =  pow(val,mut_pow) - 1.0;
            }
            else
            {
                xy = 1.0-delta2;
                val = 2.0*(1.0-rnd)+2.0*(rnd-0.5)*(pow(xy,(eta_m+1.0)));
                deltaq = 1.0 - (pow(val,mut_pow));
            }
            y = y + deltaq*(yu-yl);
            if (y<yl)          y = yl;
                               if (y>yu)          y = yu;
                               ind->xreal[j] = y;
                               nrealmut+=1;
                    }
        }
            return;
}

/* Test problem definitions */

# include <stdio.h>
# include <stdlib.h>
# include <math.h>

# include "global.h"
# include "rand.h"

/* # define sch1 */
/* # define sch2 */
/* # define fon */
/* # define kur */
/* # define pol */
/* # define vnt */
/* # define zdt1*/
/* # define zdt2 */
/* # define zdt3 */
/* # define zdt4 */
/* # define zdt5 */
/* # define zdt6 */
/* # define bnh */
```

```
/* # define osy */
/* # define srn */
/* # define tnk */
/* # define ctp1 */
/* # define ctp2 */
/* # define ctp3 */
/* # define ctp4 */
/* # define ctp5 */
/* # define ctp6 */
/* # define ctp7 */
/* # define ctp8 */
#define generator_matchup
/*  Test problem SCH1
   # of real variables = 1
   # of bin variables = 0
   # of objectives = 2
   # of constraints = 0
   */

#ifdef sch1
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   obj[0] = pow(xreal[0],2.0);
   obj[1] = pow((xreal[0]-2.0),2.0);
   return;
}
#endif

/*  Test problem SCH2
   # of real variables = 1
   # of bin variables = 0
   # of objectives = 2
   # of constraints = 0
   */

#ifdef sch2
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   if (xreal[0]<=1.0)
   {
      obj[0] = -xreal[0];
      obj[1] = pow((xreal[0]-5.0),2.0);
      return;
   }
   if (xreal[0]<=3.0)
   {
```

```
        obj[0] = xreal[0]-2.0;
        obj[1] = pow((xreal[0]-5.0),2.0);
        return;
    }
    if (xreal[0]<=4.0)
    {
        obj[0] = 4.0-xreal[0];
        obj[1] = pow((xreal[0]-5.0),2.0);
        return;
    }
    obj[0] = xreal[0]-4.0;
    obj[1] = pow((xreal[0]-5.0),2.0);
    return;
}
#endif

/*  Test problem FON
    # of real variables = n
    # of bin variables = 0
    # of objectives = 2
    # of constraints = 0
    */

#ifdef fon
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
    double s1, s2;
    int i;
    s1 = s2 = 0.0;
    for (i=0; i<nreal; i++)
    {
        s1 += pow((xreal[i]-(1.0/sqrt((double)nreal))),2.0);
        s2 += pow((xreal[i]+(1.0/sqrt((double)nreal))),2.0);
    }
    obj[0] = 1.0 - exp(-s1);
    obj[1] = 1.0 - exp(-s2);
    return;
}
#endif

/*  Test problem KUR
    # of real variables = 3
    # of bin variables = 0
    # of objectives = 2
    # of constraints = 0
    */
```

```
#ifdef kur
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   int i;
   double res1, res2;
   res1 = -0.2*sqrt((xreal[0]*xreal[0]) + (xreal[1]*xreal[1]));
   res2 = -0.2*sqrt((xreal[1]*xreal[1]) + (xreal[2]*xreal[2]));
   obj[0] = -10.0*( exp(res1) + exp(res2));
   obj[1] = 0.0;
   for (i=0; i<3; i++)
   {
      obj[1] += pow(fabs(xreal[i]),0.8) + 5.0*sin(pow(xreal[i],3.0));
   }
   return;
}
#endif

/*  Test problem POL
   # of real variables = 2
   # of bin variables = 0
   # of objectives = 2
   # of constraints = 0
   */

#ifdef pol
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   double a1, a2, b1, b2;
   a1 = 0.5*sin(1.0) - 2.0*cos(1.0) + sin(2.0) - 1.5*cos(2.0);
   a2 = 1.5*sin(1.0) - cos(1.0) + 2.0*sin(2.0) - 0.5*cos(2.0);
   b1 = 0.5*sin(xreal[0]) - 2.0*cos(xreal[0]) + sin(xreal[1]) - 1.5*cos(xreal[1]);
   b2 = 1.5*sin(xreal[0]) - cos(xreal[0]) + 2.0*sin(xreal[1]) - 0.5*cos(xreal[1]);
   obj[0] = 1.0 + pow((a1-b1),2.0) + pow((a2-b2),2.0);
   obj[1] = pow((xreal[0]+3.0),2.0) + pow((xreal[1]+1.0),2.0);
   return;
}
#endif

/*  Test problem VNT
   # of real variables = 2
   # of bin variables = 0
   # of objectives = 3
   # of constraints = 0
   */
```

```
#ifdef vnt
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   obj[0] = 0.5*(xreal[0]*xreal[0] + xreal[1]*xreal[1]) + sin(xreal[0]*xreal[0] +
xreal[1]*xreal[1]);
   obj[1] = (pow((3.0*xreal[0] - 2.0*xreal[1] + 4.0),2.0))/8.0 + (pow((xreal[0]-
xreal[1]+1.0),2.0))/27.0 + 15.0;
   obj[2] = 1.0/(xreal[0]*xreal[0] + xreal[1]*xreal[1] + 1.0) - 1.1*exp(-(xreal[0]*xreal[0] +
xreal[1]*xreal[1]));
   return;
}
#endif

/*  Test problem ZDT1
   # of real variables = 30
   # of bin variables = 0
   # of objectives = 2
   # of constraints = 0
   */

#ifdef zdt1
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   double f1, f2, g, h;
   int i;
   f1 = xreal[0];
   g = 0.0;
   for (i=1; i<30; i++)
   {
      g += xreal[i];
   }
   g = 9.0*g/29.0;
   g += 1.0;
   h = 1.0 - sqrt(f1/g);
   f2 = g*h;
   obj[0] = f1;
   obj[1] = f2;
   return;
}
#endif

/*  Test problem ZDT2
   # of real variables = 30
   # of bin variables = 0
   # of objectives = 2
   # of constraints = 0
```

```
   */

#ifdef zdt2
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   double f1, f2, g, h;
   int i;
   f1 = xreal[0];
   g = 0.0;
   for (i=1; i<30; i++)
   {
      g += xreal[i];
   }
   g = 9.0*g/29.0;
   g += 1.0;
   h = 1.0 - pow((f1/g),2.0);
   f2 = g*h;
   obj[0] = f1;
   obj[1] = f2;
   return;
}
#endif

/*  Test problem ZDT3
   # of real variables = 30
   # of bin variables = 0
   # of objectives = 2
   # of constraints = 0
   */

#ifdef zdt3
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   double f1, f2, g, h;
   int i;
   f1 = xreal[0];
   g = 0.0;
   for (i=1; i<30; i++)
   {
      g += xreal[i];
   }
   g = 9.0*g/29.0;
   g += 1.0;
   h = 1.0 - sqrt(f1/g) - (f1/g)*sin(10.0*PI*f1);
   f2 = g*h;
   obj[0] = f1;
```

```
   obj[1] = f2;
   return;
}
#endif

/*  Test problem ZDT4
   # of real variables = 10
   # of bin variables = 0
   # of objectives = 2
   # of constraints = 0
   */

#ifdef zdt4
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   double f1, f2, g, h;
   int i;
   f1 = xreal[0];
   g = 0.0;
   for (i=1; i<10; i++)
   {
      g += xreal[i]*xreal[i] - 10.0*cos(4.0*PI*xreal[i]);
   }
   g += 91.0;
   h = 1.0 - sqrt(f1/g);
   f2 = g*h;
   obj[0] = f1;
   obj[1] = f2;
   return;
}
#endif

/*  Test problem ZDT5
   # of real variables = 0
   # of bin variables = 11
   # of bits for binvar1 = 30
   # of bits for binvar2-11 = 5
   # of objectives = 2
   # of constraints = 0
   */

#ifdef zdt5
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   int i, j;
   int u[11];
```

```
    int v[11];
    double f1, f2, g, h;
    for (i=0; i<11; i++)
    {
       u[i] = 0;
    }
    for (j=0; j<30; j++)
    {
       if (gene[0][j] == 1)
       {
          u[0]++;
       }
    }
    for (i=1; i<11; i++)
    {
       for (j=0; j<4; j++)
       {
          if (gene[i][j] == 1)
          {
             u[i]++;
          }
       }
    }
    f1 = 1.0 + u[0];
    for (i=1; i<11; i++)
    {
       if (u[i] < 5)
       {
          v[i] = 2 + u[i];
       }
       else
       {
          v[i] = 1;
       }
    }
    g = 0;
    for (i=1; i<11; i++)
    {
       g += v[i];
    }
    h = 1.0/f1;
    f2 = g*h;
    obj[0] = f1;
    obj[1] = f2;
    return;
}
```

```
#endif

/*  Test problem ZDT6
    # of real variables = 10
    # of bin variables = 0
    # of objectives = 2
    # of constraints = 0
    */

#ifdef zdt6
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
    double f1, f2, g, h;
    int i;
    f1 = 1.0 - (exp(-4.0*xreal[0]))*pow((sin(4.0*PI*xreal[0])),6.0);
    g = 0.0;
    for (i=1; i<10; i++)
    {
        g += xreal[i];
    }
    g = g/9.0;
    g = pow(g,0.25);
    g = 1.0 + 9.0*g;
    h = 1.0 - pow((f1/g),2.0);
    f2 = g*h;
    obj[0] = f1;
    obj[1] = f2;
    return;
}
#endif

/*  Test problem BNH
    # of real variables = 2
    # of bin variables = 0
    # of objectives = 2
    # of constraints = 2
    */

#ifdef bnh
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
    obj[0] = 4.0*(xreal[0]*xreal[0] + xreal[1]*xreal[1]);
    obj[1] = pow((xreal[0]-5.0),2.0) + pow((xreal[1]-5.0),2.0);
    constr[0] = 1.0 - (pow((xreal[0]-5.0),2.0) + xreal[1]*xreal[1])/25.0;
    constr[1] = (pow((xreal[0]-8.0),2.0) + pow((xreal[1]+3.0),2.0))/7.7 - 1.0;
    return;
```

```
}
#endif

/*  Test problem OSY
   # of real variables = 6
   # of bin variables = 0
   # of objectives = 2
   # of constraints = 6
   */

#ifdef osy
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   obj[0] = -(25.0*pow((xreal[0]-2.0),2.0) + pow((xreal[1]-2.0),2.0) + pow((xreal[2]-1.0),2.0) +
pow((xreal[3]-4.0),2.0) + pow((xreal[4]-1.0),2.0));
   obj[1] = xreal[0]*xreal[0] +  xreal[1]*xreal[1] + xreal[2]*xreal[2] + xreal[3]*xreal[3] +
xreal[4]*xreal[4] + xreal[5]*xreal[5];
   constr[0] = (xreal[0]+xreal[1])/2.0 - 1.0;
   constr[1] = 1.0 - (xreal[0]+xreal[1])/6.0;
   constr[2] = 1.0 - xreal[1]/2.0 + xreal[0]/2.0;
   constr[3] = 1.0 - xreal[0]/2.0 + 3.0*xreal[1]/2.0;
   constr[4] = 1.0 - (pow((xreal[2]-3.0),2.0))/4.0 - xreal[3]/4.0;
   constr[5] = (pow((xreal[4]-3.0),2.0))/4.0 + xreal[5]/4.0 - 1.0;
   return;
}
#endif

/*  Test problem SRN
   # of real variables = 2
   # of bin variables = 0
   # of objectives = 2
   # of constraints = 2
   */

#ifdef srn
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   obj[0] = 2.0 + pow((xreal[0]-2.0),2.0) + pow((xreal[1]-1.0),2.0);
   obj[1] = 9.0*xreal[0] - pow((xreal[1]-1.0),2.0);
   constr[0] = 1.0 - (pow(xreal[0],2.0) + pow(xreal[1],2.0))/225.0;
   constr[1] = 3.0*xreal[1]/10.0 - xreal[0]/10.0 - 1.0;
   return;
}
#endif

/*  Test problem TNK
```

```
   # of real variables = 2
   # of bin variables = 0
   # of objectives = 2
   # of constraints = 2
   */

#ifdef tnk
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   obj[0] = xreal[0];
   obj[1] = xreal[1];
   if (xreal[1] == 0.0)
   {
      constr[0] = -1.0;
   }
   else
   {
      constr[0] = xreal[0]*xreal[0] + xreal[1]*xreal[1] - 0.1*cos(16.0*atan(xreal[0]/xreal[1])) -
1.0;
   }
   constr[1] = 1.0 - 2.0*pow((xreal[0]-0.5),2.0) + 2.0*pow((xreal[1]-0.5),2.0);
   return;
}
#endif

/*  Test problem CTP1
   # of real variables = 2
   # of bin variables = 0
   # of objectives = 2
   # of constraints = 2
   */

#ifdef ctp1
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   double g;
   g = 1.0 + xreal[1];
   obj[0] = xreal[0];
   obj[1] = g*exp(-obj[0]/g);
   constr[0] = obj[1]/(0.858*exp(-0.541*obj[0]))-1.0;
   constr[1] = obj[1]/(0.728*exp(-0.295*obj[0]))-1.0;
   return;
}
#endif

/*  Test problem CTP2
```

```
   # of real variables = 2
   # of bin variables = 0
   # of objectives = 2
   # of constraints = 1
   */

#ifdef ctp2
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   double g;
   double theta, a, b, c, d, e;
   double exp1, exp2;
   theta = -0.2*PI;
   a = 0.2;
   b = 10.0;
   c = 1.0;
   d = 6.0;
   e = 1.0;
   g = 1.0 + xreal[1];
   obj[0] = xreal[0];
   obj[1] = g*(1.0  - sqrt(obj[0]/g));
   exp1 = (obj[1]-e)*cos(theta) - obj[0]*sin(theta);
   exp2 = (obj[1]-e)*sin(theta) + obj[0]*cos(theta);
   exp2 = b*PI*pow(exp2,c);
   exp2 = fabs(sin(exp2));
   exp2 = a*pow(exp2,d);
   constr[0] = exp1/exp2 - 1.0;
   return;
}
#endif

/*  Test problem CTP3
   # of real variables = 2
   # of bin variables = 0
   # of objectives = 2
   # of constraints = 1
   */

#ifdef ctp3
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   double g;
   double theta, a, b, c, d, e;
   double exp1, exp2;
   theta = -0.2*PI;
   a = 0.1;
```

```
    b = 10.0;
    c = 1.0;
    d = 0.5;
    e = 1.0;
    g = 1.0 + xreal[1];
    obj[0] = xreal[0];
    obj[1] = g*(1.0  - sqrt(obj[0]/g));
    exp1 = (obj[1]-e)*cos(theta) - obj[0]*sin(theta);
    exp2 = (obj[1]-e)*sin(theta) + obj[0]*cos(theta);
    exp2 = b*PI*pow(exp2,c);
    exp2 = fabs(sin(exp2));
    exp2 = a*pow(exp2,d);
    constr[0] = exp1/exp2 - 1.0;
    return;
}
#endif

/*  Test problem CTP4
    # of real variables = 2
    # of bin variables = 0
    # of objectives = 2
    # of constraints = 1
    */

#ifdef ctp4
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
    double g;
    double theta, a, b, c, d, e;
    double exp1, exp2;
    theta = -0.2*PI;
    a = 0.75;
    b = 10.0;
    c = 1.0;
    d = 0.5;
    e = 1.0;
    g = 1.0 + xreal[1];
    obj[0] = xreal[0];
    obj[1] = g*(1.0  - sqrt(obj[0]/g));
    exp1 = (obj[1]-e)*cos(theta) - obj[0]*sin(theta);
    exp2 = (obj[1]-e)*sin(theta) + obj[0]*cos(theta);
    exp2 = b*PI*pow(exp2,c);
    exp2 = fabs(sin(exp2));
    exp2 = a*pow(exp2,d);
    constr[0] = exp1/exp2 - 1.0;
    return;
```

```
}
#endif

/*  Test problem CTP5
   # of real variables = 2
   # of bin variables = 0
   # of objectives = 2
   # of constraints = 1
   */

#ifdef ctp5
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   double g;
   double theta, a, b, c, d, e;
   double exp1, exp2;
   theta = -0.2*PI;
   a = 0.1;
   b = 10.0;
   c = 2.0;
   d = 0.5;
   e = 1.0;
   g = 1.0 + xreal[1];
   obj[0] = xreal[0];
   obj[1] = g*(1.0  - sqrt(obj[0]/g));
   exp1 = (obj[1]-e)*cos(theta) - obj[0]*sin(theta);
   exp2 = (obj[1]-e)*sin(theta) + obj[0]*cos(theta);
   exp2 = b*PI*pow(exp2,c);
   exp2 = fabs(sin(exp2));
   exp2 = a*pow(exp2,d);
   constr[0] = exp1/exp2 - 1.0;
   return;
}
#endif

/*  Test problem CTP6
   # of real variables = 2
   # of bin variables = 0
   # of objectives = 2
   # of constraints = 1
   */

#ifdef ctp6
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   double g;
```

```
    double theta, a, b, c, d, e;
    double exp1, exp2;
    theta = 0.1*PI;
    a = 40.0;
    b = 0.5;
    c = 1.0;
    d = 2.0;
    e = -2.0;
    g = 1.0 + xreal[1];
    obj[0] = xreal[0];
    obj[1] = g*(1.0  - sqrt(obj[0]/g));
    exp1 = (obj[1]-e)*cos(theta) - obj[0]*sin(theta);
    exp2 = (obj[1]-e)*sin(theta) + obj[0]*cos(theta);
    exp2 = b*PI*pow(exp2,c);
    exp2 = fabs(sin(exp2));
    exp2 = a*pow(exp2,d);
    constr[0] = exp1/exp2 - 1.0;
    return;
}
#endif

/*  Test problem CTP7
    # of real variables = 2
    # of bin variables = 0
    # of objectives = 2
    # of constraints = 1
    */

#ifdef ctp7
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
    double g;
    double theta, a, b, c, d, e;
    double exp1, exp2;
    theta = -0.05*PI;
    a = 40.0;
    b = 5.0;
    c = 1.0;
    d = 6.0;
    e = 0.0;
    g = 1.0 + xreal[1];
    obj[0] = xreal[0];
    obj[1] = g*(1.0  - sqrt(obj[0]/g));
    exp1 = (obj[1]-e)*cos(theta) - obj[0]*sin(theta);
    exp2 = (obj[1]-e)*sin(theta) + obj[0]*cos(theta);
    exp2 = b*PI*pow(exp2,c);
```

```
    exp2 = fabs(sin(exp2));
    exp2 = a*pow(exp2,d);
    constr[0] = exp1/exp2 - 1.0;
    return;
}
#endif

/*  Test problem CTP8
    # of real variables = 2
    # of bin variables = 0
    # of objectives = 2
    # of constraints = 2
    */

#ifdef ctp8
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
    double g;
    double theta, a, b, c, d, e;
    double exp1, exp2;
    g = 1.0 + xreal[1];
    obj[0] = xreal[0];
    obj[1] = g*(1.0  - sqrt(obj[0]/g));
    theta = 0.1*PI;
    a = 40.0;
    b = 0.5;
    c = 1.0;
    d = 2.0;
    e = -2.0;
    exp1 = (obj[1]-e)*cos(theta) - obj[0]*sin(theta);
    exp2 = (obj[1]-e)*sin(theta) + obj[0]*cos(theta);
    exp2 = b*PI*pow(exp2,c);
    exp2 = fabs(sin(exp2));
    exp2 = a*pow(exp2,d);
    constr[0] = exp1/exp2 - 1.0;
    theta = -0.05*PI;
    a = 40.0;
    b = 2.0;
    c = 1.0;
    d = 6.0;
    e = 0.0;
    exp1 = (obj[1]-e)*cos(theta) - obj[0]*sin(theta);
    exp2 = (obj[1]-e)*sin(theta) + obj[0]*cos(theta);
    exp2 = b*PI*pow(exp2,c);
    exp2 = fabs(sin(exp2));
    exp2 = a*pow(exp2,d);
```

```
   constr[1] = exp1/exp2 - 1.0;
   return;
}
#endif

#ifdef generator_matchup
void test_problem (double **xreal,double *gen_cost, double *load_cost, double **load, double
*max_realvar, int num_loads, int nreal, int *group, int group_max, double *obj, double *constr)
{
  int i,j,k,l, hrs=24;
  obj[0] =0.0;
  obj[1] = 0.0;
double sum =0.0, sum1=0.0;
/* Objective function definition */

   for (i = 0; i < hrs ; i++)
       {
       for ( j = 0; j < nreal ; j++)
           {
           obj[0] += (gen_cost[j]*xreal[j][i];
           }
       for ( k= 0; k < num_loads ; k++)
           {
           obj[0] -= load_cost[k]*load[k][i];
           }
       }
   for (i = 0; i < hrs ; i++)
       {
       for ( j = 0; j < nreal ; j++)
           {
           for( l = 0; l < group_max ; l++)
              {
              if ( j == group[l])
                sum1 += max_realvar[j] - xreal[j][i];
              }
           sum += max_realvar[j] -x real[j][i];
           }
        for ( j = 0; j < nreal ; j++)
           {
           for( l = 0; l < group_max ; l++)
              {
              if ( j != group[l])
                 obj[1] += ((max_realvar[j]- xreal[j][i])/sum)^2;
              }
           }
         obj[1] += (sum1/sum)^2;
```

```
        }
       obj[1] = obj[1]/24;
/* constraints definition */
/* Economic Minimum and Maximum Operating Cosntaraint *

    for (i = 0; i < hrs ; i++)
        {
        for ( j = 0; j < nreal ; j++)
            {
            constr[j]=max_realvar[j] -(xreal[j][i]+ Spin[j];
            }
        }
    for (i = 0; i < hrs ; i++)
        {
        for ( j = 0; j < nreal ; j++)
            {
            constr[j+nreal]=max_realvar[j] -(xreal[j][i]+ SOper[j];
            }
        }
    for (i = 0; i < hrs ; i++)
        {
        for ( j = 0; j < nreal ; j++)
            {
            constr[j+2*nreal]=(xreal[j][i] -min_realvar[j];
            }
        }


    return;
}
#endif

/* Definition of random number generation routines */

# include <stdio.h>
# include <stdlib.h>
# include <math.h>

# include "global.h"
# include "rand.h"

double seed;
double oldrand[55];
int jrand;

/* Get seed number for random and start it up */
```

```
void randomize()
{
   int j1;
   for(j1=0; j1<=54; j1++)
   {
      oldrand[j1] = 0.0;
   }
   jrand=0;
   warmup_random (seed);
   return;
}

/* Get randomize off and running */
void warmup_random (double seed)
{
   int j1, ii;
   double new_random, prev_random;
   oldrand[54] = seed;
   new_random = 0.000000001;
   prev_random = seed;
   for(j1=1; j1<=54; j1++)
   {
      ii = (21*j1)%54;
      oldrand[ii] = new_random;
      new_random = prev_random-new_random;
      if(new_random<0.0)
      {
         new_random += 1.0;
      }
      prev_random = oldrand[ii];
   }
   advance_random ();
   advance_random ();
   advance_random ();
   jrand = 0;
   return;
}

/* Create next batch of 55 random numbers */
void advance_random ()
{
   int j1;
   double new_random;
   for(j1=0; j1<24; j1++)
   {
      new_random = oldrand[j1]-oldrand[j1+31];
```

```
      if(new_random<0.0)
      {
         new_random = new_random+1.0;
      }
      oldrand[j1] = new_random;
   }
   for(j1=24; j1<55; j1++)
   {
      new_random = oldrand[j1]-oldrand[j1-24];
      if(new_random<0.0)
      {
         new_random = new_random+1.0;
      }
      oldrand[j1] = new_random;
   }
}

/* Fetch a single random number between 0.0 and 1.0 */
double randomperc()
{
   jrand++;
   if(jrand>=55)
   {
      jrand = 1;
      advance_random();
   }
   return((double)oldrand[jrand]);
}

/* Fetch a single random integer between low and high including the bounds */
int rnd (int low, int high)
{
   int res;
   if (low >= high)
   {
      res = low;
   }
   else
   {
      res = low + (randomperc()*(high-low+1));
      if (res > high)
      {
         res = high;
      }
   }
   return (res);
```

```
}

/* Fetch a single random real number between low and high including the bounds */
double rndreal (double low, double high)
{
   return (low + (high-low)*randomperc());
}

/* Declaration for random number related variables and routines */

# ifndef _RAND_H_
# define _RAND_H_

/* Variable declarations for the random number generator */
extern double seed;
extern double oldrand[55];
extern int jrand;

/* Function declarations for the random number generator */
void randomize(void);
void warmup_random (double seed);
void advance_random (void);
double randomperc(void);
int rnd (int low, int high);
double rndreal (double low, double high);

# endif

/* Routines for storing population data into files */

# include <stdio.h>
# include <stdlib.h>
# include <math.h>

# include "global.h"
# include "rand.h"

/* Function to print the information of a population in a file */
void report_pop (individual *ind, FILE *fpt)
{
   int i, j, k;
   for (i=0; i<popsize; i++)
   {
      for (j=0; j<nobj; j++)
      {
         fprintf(fpt,"%e\t",ind[i].obj[j]);
```

```
        }
        if (ncon!=0)
        {
            for (j=0; j<ncon; j++)
            {
                fprintf(fpt,"%e\t",ind[i].constr[j]);
            }
        }
        if (nreal!=0)
        {
            for (j=0; j<nbits; j++)
            {
                for ( hr <0; hr <24; hr++)
                fprintf(fpt,"%e\t",cur->ind->xreal[j][hr]);
                fprintf(fpt,"\n");
            }
        }
        if (nbits!=0)
        {

                for (k=0; k<nbits; k++)
                {
                    fprintf(fpt,"%d\t",ind[i].gene[k]);
                }

        }
        fprintf(fpt,"%e\n",ind[i].constr_violation);
    }
    return;
}

/* Function to print the information of feasible and non-dominated population in a file */
void report_archive (ind_list *elite, FILE *fpt)
{
    int j, k;
    ind_list *cur;
    cur = elite->child;
    while (cur!=NULL)
    {
        for (j=0; j<nobj; j++)
        {
            fprintf(fpt,"%e\t",cur->ind->obj[j]);
        }
        if (ncon!=0)
        {
            for (j=0; j<ncon; j++)
```

```
            {
               fprintf(fpt,"%e\t",cur->ind->constr[j]);
            }
         }
         if (nbits!=0)
         {
            for (j=0; j<nbits; j++)
            {
             for ( hr <0; hr <24; hr++)
               fprintf(fpt,"%e\t",cur->ind->xreal[j][hr]);
               fprintf(fpt,"\n");
            }
         }
         if (nbits!=0)
         {

               for (k=0; k<nbits; k++)
               {
                  fprintf(fpt,"%d\t",cur->ind->gene[k]);
               }

         }
         fprintf(fpt,"%e\n",cur->ind->constr_violation);
         cur = cur->child;
      }
      return;
}

/* Tournamenet Selections routine */

# include <stdio.h>
# include <stdlib.h>
# include <math.h>

# include "global.h"
# include "rand.h"

/* Routine for binary neigborhood */
individual* tournament (individual *ind1, individual *ind2)
{
   int flag;
   flag = check_dominance (ind1, ind2);
   if (flag==1)
   {
      return (ind1);
   }
```

```
    if (flag==-1)
    {
        return (ind2);
    }
    if ((randomperc()) <= 0.5)
    {
        return(ind1);
    }
    else
    {
        return(ind2);
    }
}

/* Routines for updating elite and EA populations */

# include <stdio.h>
# include <stdlib.h>
# include <math.h>

# include "global.h"
# include "rand.h"

/* Routine to update archive */
void update_elite (ind_list *elite, individual *ind)
{
    int i, end, flag;
    double d1, d2;
    ind_list *temp;
    temp = elite->child;
    end = 0;
    do
    {
        flag = check_box_dominance (ind, temp->ind);
        switch (flag)
        {
        case 1: /* ind dominates temp->ind */
            {
                temp = del (temp);
                temp = temp->child;
                break;
            }
        case 2: /* temp->ind dominates ind */
            {
                return;
            }
```

```
    case 3: /* both are non-dominated and are in different boxes */
        {
            temp = temp->child;
            break;
        }
    case 4: /* both are non-dominated and are in same hyper-box */
        {
            end = 1;
            break;
        }
    }
}
while (end!=1 && temp!=NULL);
if (end==0)
{
    insert(elite, ind);
}
else
{
    if (flag==4)        /* in same hyperbox */
    {
        flag = check_dominance (ind, temp->ind);
        switch (flag)
        {
        case 1:
            {
                temp = del(temp);
                insert (elite, ind);
                break;
            }
        case -1:
            {
                return;
            }
        case 0:
            {
                d1 = 0.0;
                d2 = 0.0;
                for (i=0; i<nobj; i++)
                {
                    d1 += pow(((ind->obj[i]-ind->ia[i])/epsilon[i]),2.0);
                    d2 += pow(((temp->ind->obj[i]-temp->ind->ia[i])/epsilon[i]),2.0);
                }
                if (d1<=d2)
                {
                    temp = del(temp);
```

```
                    insert(elite,ind);
                }
                break;
            }
        }
    }
  }
  return;
}

/* Routine to update population */
void update_pop (individual *ea, individual *ind)
{
        int size;
        int i;
        int flag;
        size = 0;
        for (i=0; i<popsize; i++)
        {
                flag = check_dominance (ind, &ea[i]);
                switch (flag)
                {
                        case 1:
                                copy (ind, &ea[i]);
                                return;
                        case -1:
                                return;
                        case 0:
                                array[size++] = i;
                                break;
                }
        }
        if (size>0)
        {
                i = rnd(0,size-1);
                copy (ind, &ea[array[i]]);
        }
        return;
}
```

NSGA II Algorithm

```
/* Memory allocation and deallocation routines */
# include <stdio.h>
# include <stdlib.h>
# include <math.h>
```

```
# include "global.h"
# include "rand.h"
/* Function to allocate memory to variables of an individual */
void allocate (individual *ind)
{
   int j;
   if (nbits != 0)
   {
      ind->xreal = (double **)malloc(nbits*sizeof(double));
      ind->gen_cost = (double **)malloc(nbits*sizeof(double));
      ind->gene = (int *)malloc(nbits*sizeof(int));
      for (hr=0; hr<24; j++)
      {
        ind->xreal[hr] = (double *)malloc(nbits*sizeof(double));
        ind->gen_cost[hr] = (double *)malloc(nbits*sizeof(double))
      }
   }

   ind->obj = (double *)malloc(nobj*sizeof(double));
   ind->ia = (int *)malloc(nobj*sizeof(int));
   if (ncon != 0)
   {
      ind->constr = (double *)malloc(ncon*sizeof(double));
   }
   right_genes = (double *)malloc((nbits-site1)*sizeof(double));
   max_gen = (double *)malloc((nbits-site1)*sizeof(double));
   assinged = (double *)malloc(nbits*sizeof(double));
   return;
}

/* Function to deallocate memory of variables of an individual */
void deallocate (individual *ind)
{
   int j;
   if (nbits != 0)
   {
      for (hr=0; hr<24; j++)
      {
       free(ind->xreal[hr]);
       free(ind->gen_cost[hr]);
      }
      free(ind->xreal);
      free(ind->gen_cost);

      free(ind->gene);
```

```
    }
    free(ind->obj);
    free(ind->ia);
    if (ncon != 0)
    {
        free(ind->constr);
    }
    free(right_genes);
    free(max_gen);
    free(assigned);
    return;
}


/* Routine for mergeing two populations */

# include <stdio.h>
# include <stdlib.h>
# include <math.h>

# include "global.h"
# include "rand.h"

/* Routine to copy an individual 'ind1' into another individual 'ind2' */
void copy (individual *ind1, individual *ind2)
{
    int i, j;
    ind2->constr_violation = ind1->constr_violation;
    if (nreal!=0)
    {
        for (i=0; i<nreal; i++)
        {
            ind2->xreal[i] = ind1->xreal[i];
        }
    }
    if (nbits!=0)
    {

            for (j=0; j<nbits; j++)
            {
                ind2->gene[j] = ind1->gene[j];
                for ( hr =0; hr <24; hr++)
                    ind2->xreal[j][hr] = ind1->xreal[j][hr];
                    ind2->gen_cost[j][hr] = ind1->gen_cost[j];
            }
```

```
    }
    for (i=0; i<nobj; i++)
    {
        ind2->obj[i] = ind1->obj[i];
        ind2->ia[i] = ind1->ia[i];
    }
    if (ncon!=0)
    {
        for (i=0; i<ncon; i++)
        {
            ind2->constr[i] = ind1->constr[i];
        }
    }
    return;
}
/* Crossover routines */

# include <stdio.h>
# include <stdlib.h>
# include <math.h>

# include "global.h"
# include "rand.h"

/* Function to cross two individuals */
void crossover (individual *parent1, individual *parent2, individual *child1, individual *child2)
{

    if (nbits!=0)
    {
        bincross (parent1, parent2, child1, child2);
    }
    return;
}


/* Routine for single point binary crossover */
void bincross (individual *parent1, individual *parent2, individual *child1, individual *child2)
{
    int i, j;
    double rand;
    int temp, site1;

        rand = randomperc();
        if (rand <= pcross_bin)
```

```
        {
           nbincross++;

           for (j=0; j<site1; j++)
           {
              child1->gene[j] = parent1->gene[j];
              child2->gene[j] = parent2->gene[j];
              for ( hr= 0 ; hr < 24; hr++)
                  {
                  child1->xreal[j][hr]= parent1->xreal[j][hr];
                  child2->xreal[j][hr]= parent2->xreal[j][hr];
                  }
           }

           for (j=site1; j<nbits; j++)
           {
              child1->gene[j] = parent2->gene[j];
              child2->gene[j] = parent1->gene[j];
              for ( hr= 0 ; hr < 24; hr++)
                  {
                  child1->xreal[j][hr]= parent2->xreal[j][hr];
                  child2->xreal[j][hr]= parent1->xreal[j][hr];
                  }
           }
        }
        else
        {
           for (j=0; j<nbits; j++)
           {
              child1->gene[j] = parent1->gene[j];
              child2->gene[j] = parent2->gene[j];
              for ( hr= 0 ; hr < 24; hr++)
                  {
                  child1->xreal[j][hr]= parent1->xreal[j][hr];
                  child2->xreal[j][hr]= parent2->xreal[j][hr];
                  }
           }
        }

    return;
}

/* Crowding distance computation routines */

# include <stdio.h>
# include <stdlib.h>
```

```
# include <math.h>

# include "global.h"
# include "rand.h"

/* Routine to compute crowding distance based on ojbective function values when the population
in in the form of a list */
void assign_crowding_distance_list (population *pop, list *lst, int front_size)
{
   int **obj_array;
   int *dist;
   int i, j;
   list *temp;
   temp = lst;
   if (front_size==1)
   {
      pop->ind[lst->index].crowd_dist = INF;
      return;
   }
   if (front_size==2)
   {
      pop->ind[lst->index].crowd_dist = INF;
      pop->ind[lst->child->index].crowd_dist = INF;
      return;
   }
   obj_array = (int **)malloc(nobj*sizeof(int));
   dist = (int *)malloc(front_size*sizeof(int));
   for (i=0; i<nobj; i++)
   {
      obj_array[i] = (int *)malloc(front_size*sizeof(int));
   }
   for (j=0; j<front_size; j++)
   {
      dist[j] = temp->index;
      temp = temp->child;
   }
   assign_crowding_distance (pop, dist, obj_array, front_size);
   free (dist);
   for (i=0; i<nobj; i++)
   {
      free (obj_array[i]);
   }
   free (obj_array);
   return;
}
```

```
/* Routine to compute crowding distance based on objective function values when the population
in in the form of an array */
void assign_crowding_distance_indices (population *pop, int c1, int c2)
{
   int **obj_array;
   int *dist;
   int i, j;
   int front_size;
   front_size = c2-c1+1;
   if (front_size==1)
   {
      pop->ind[c1].crowd_dist = INF;
      return;
   }
   if (front_size==2)
   {
      pop->ind[c1].crowd_dist = INF;
      pop->ind[c2].crowd_dist = INF;
      return;
   }
   obj_array = (int **)malloc(nobj*sizeof(int));
   dist = (int *)malloc(front_size*sizeof(int));
   for (i=0; i<nobj; i++)
   {
      obj_array[i] = (int *)malloc(front_size*sizeof(int));
   }
   for (j=0; j<front_size; j++)
   {
      dist[j] = c1++;
   }
   assign_crowding_distance (pop, dist, obj_array, front_size);
   free (dist);
   for (i=0; i<nobj; i++)
   {
      free (obj_array[i]);
   }
   free (obj_array);
   return;
}

/* Routine to compute crowding distances */
void assign_crowding_distance (population *pop, int *dist, int **obj_array, int front_size)
{
   int i, j;
   for (i=0; i<nobj; i++)
   {
```

```
        for (j=0; j<front_size; j++)
        {
            obj_array[i][j] = dist[j];
        }
        quicksort_front_obj (pop, i, obj_array[i], front_size);
    }
    for (j=0; j<front_size; j++)
    {
        pop->ind[dist[j]].crowd_dist = 0.0;
    }
    for (i=0; i<nobj; i++)
    {
        pop->ind[obj_array[i][0]].crowd_dist = INF;
    }
    for (i=0; i<nobj; i++)
    {
        for (j=1; j<front_size-1; j++)
        {
            if (pop->ind[obj_array[i][j]].crowd_dist != INF)
            {
                if (pop->ind[obj_array[i][front_size-1]].obj[i] == pop->ind[obj_array[i][0]].obj[i])
                {
                    pop->ind[obj_array[i][j]].crowd_dist += 0.0;
                }
                else
                {
                    pop->ind[obj_array[i][j]].crowd_dist += (pop->ind[obj_array[i][j+1]].obj[i] - pop-
>ind[obj_array[i][j-1]].obj[i])/(pop->ind[obj_array[i][front_size-1]].obj[i] - pop-
>ind[obj_array[i][0]].obj[i]);
                }
            }
        }
    }
    for (j=0; j<front_size; j++)
    {
        if (pop->ind[dist[j]].crowd_dist != INF)
        {
            pop->ind[dist[j]].crowd_dist = (pop->ind[dist[j]].crowd_dist)/nobj;
        }
    }
    return;
}

/* Domination checking routines */

# include <stdio.h>
```

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

# include "global.h"
# include "rand.h"

/*      It returns the following
        1 if a dominates b
        2 if b dominates a
        3 if a and b are non-dominated and a!=b (identification arrays unequal)
        4 if a and b are non-dominated and a=b */

int check_box_dominance (individual *a, individual *b)
{
   int i;
   int flag1;
   int flag2;
   flag1 = 0;
   flag2 = 0;
   if (a->constr_violation<0.0 && b->constr_violation<0.0)
   {
      if (a->constr_violation > b->constr_violation)
      {
         return (1);
      }
      else
      {
         if (a->constr_violation < b->constr_violation)
         {
            return (2);
         }
         else
         {
            return (4);
         }
      }
   }
   else
   {
      if (a->constr_violation<0.0 && b->constr_violation==0.0)
      {
         return (2);
      }
      else
      {
         if (a->constr_violation==0.0 && b->constr_violation<0.0)
```

```
        {
            return (1);
        }
        else
        {
            for (i=0; i<nobj; i++)
            {
                if (a->ia[i] < b->ia[i])
                {
                    flag1 = 1;

                }
                else
                {
                    if (a->ia[i] > b->ia[i])
                    {
                        flag2 = 1;
                    }
                }
            }
            if (flag1==1 && flag2==0)
            {
                return (1);
            }
            else
            {
                if (flag1==0 && flag2==1)
                {
                    return (2);
                }
                else
                {
                    if (flag1==1 && flag2==1)
                    {
                        return(3);
                    }
                    else
                    {
                        return(4);
                    }
                }
            }
        }
    }
}
```

```
/* Routine for usual non-domination checking
   It will return the following values
   1 if a dominates b
   -1 if b dominates a
   0 if both a and b are non-dominated */

int check_dominance (individual *a, individual *b)
{
   int i;
   int flag1;
   int flag2;
   flag1 = 0;
   flag2 = 0;
   if (a->constr_violation<0.0 && b->constr_violation<0.0)
   {
      if (a->constr_violation > b->constr_violation)
      {
         return (1);
      }
      else
      {
         if (a->constr_violation < b->constr_violation)
         {
            return (-1);
         }
         else
         {
            return (0);
         }
      }
   }
   else
   {
      if (a->constr_violation<0.0 && b->constr_violation==0.0)
      {
         return (-1);
      }
      else
      {
         if (a->constr_violation==0.0 && b->constr_violation<0.0)
         {
            return (1);
         }
         else
         {
```

```
            for (i=0; i<nobj; i++)
            {
               if (a->obj[i] < b->obj[i])
               {
                  flag1 = 1;

               }
               else
               {
                  if (a->obj[i] > b->obj[i])
                  {
                     flag2 = 1;
                  }
               }
            }
            if (flag1==1 && flag2==0)
            {
               return (1);
            }
            else
            {
               if (flag1==0 && flag2==1)
               {
                  return (-1);
               }
               else
               {
                  return (0);
               }
            }
         }
      }
   }
}

/* Routine for evaluating population members  */

# include <stdio.h>
# include <stdlib.h>
# include <math.h>

# include "global.h"
# include "rand.h"

/* Routine to evaluate objective function values and constraints for a population */
void evaluate_pop (population *pop)
```

```
{
   int i;
   for (i=0; i<popsize; i++)
   {
      evaluate_ind (&(pop->ind[i]));
   }
   return;
}

/* Routine to evaluate objective function values and constraints for an individual */
void evaluate_ind (individual *ind)
{
   int j;
   test_problem (ind->xreal, ind->gene, ind->obj, ind->constr);
   if (ncon==0)
   {
      ind->constr_violation = 0.0;
   }
   else
   {
      ind->constr_violation = 0.0;
      for (j=0; j<ncon; j++)
      {
         if (ind->constr[j]<0.0)
         {
            ind->constr_violation += ind->constr[j];
         }
      }
   }
   return;
}

/* NSGA-II routine (implementation of the 'main' function) *//*
 *
 */
# include <stdio.h>
# include <stdlib.h>
# include <math.h>

# include "global.h"
# include "rand.h"

int nreal;
int nbin;
int nobj;
int ncon;
```

```
int popsize;
double pcross_real;
double pcross_bin;
double pmut_real;
double pmut_bin;
double eta_c;
double eta_m;
int neval;
int currenteval;
int nbinmut;
int nrealmut;
int nbincross;
int nrealcross;
int *nbits;
int *array;
double *min_realvar;
double *max_realvar;
double *min_binvar;
double *max_binvar;
double *epsilon;
double *min_obj;
int bitlength;
int elite_size;

int main (int argc, char **argv)
{
   int i;
   int index, index1, index2;
   FILE *fpt1;
   FILE *fpt2;
   FILE *fpt3;
   FILE *fpt4;
   FILE *fpt5;
   individual *ea;
   individual *parent1, *parent2, *child1, *child2;
   ind_list *elite, *cur;
   if (argc<2)
   {
      printf("\n Usage ./main random_seed \n");
      exit(1);
   }
   seed = (double)atof(argv[1]);
   if (seed<=0.0 || seed>=1.0)
   {
      printf("\n Entered seed value is wrong, seed value must be in (0,1) \n");
      exit(1);
```

```
}
fpt1 = fopen("initial_pop.out","w");
fpt2 = fopen("final_pop.out","w");
fpt3 = fopen("final_archive.out","w");
fpt4 = fopen("all_archive.out","w");
fpt5 = fopen("params.out","w");
fprintf(fpt1,"# This file contains the data of initial population\n");
fprintf(fpt2,"# This file contains the data of final population\n");
fprintf(fpt3,"# This file contains the best obtained solution(s)\n");
fprintf(fpt4,"# This file contains the data of archive for all generations\n");
fprintf(fpt5,"# This file contains information about inputs as read by the program\n");
printf("\n Enter the problem relevant and algorithm relevant parameters ... ");
printf("\n Enter the population size (>1) : ");
scanf("%d",&popsize);
if (popsize<2)
{
   printf("\n population size read is : %d",popsize);
   printf("\n Wrong population size entered, hence exiting \n");
   exit (1);
}
printf("\n Enter the number of function evaluations : ");
scanf("%d",&neval);
if (neval<popsize)
{
   printf("\n number of function evaluations read is : %d",neval);
   printf("\n Wrong nuber of evaluations entered, hence exiting \n");
   exit (1);
}
printf("\n Enter the number of objectives (>=2): ");
scanf("%d",&nobj);
if (nobj<2)
{
   printf("\n number of objectives entered is : %d",nobj);
   printf("\n Wrong number of objectives entered, hence exiting \n");
   exit (1);
}
epsilon = (double *)malloc(nobj*sizeof(double));
min_obj = (double *)malloc(nobj*sizeof(double));
for (i=0; i<nobj; i++)
{
   printf("\n Enter the value of epsilon[%d] : ",i+1);
   scanf("%lf",&epsilon[i]);
   if (epsilon[i]<=0.0)
   {
      printf("\n Entered value of epsilon[%d] is non-positive, hence exiting\n",i+1);
      exit(1);
```

```
        }
        printf("\n Enter the value of min_obj[%d] (if not known, enter 0.0) : ",i+1);
        scanf("%lf",&min_obj[i]);
    }
    printf("\n Enter the number of constraints : ");
    scanf("%d",&ncon);
    if (ncon<0)
    {
        printf("\n number of constraints entered is : %d",ncon);
        printf("\n Wrong number of constraints enetered, hence exiting \n");
        exit (1);
    }
    printf("\n Enter the number of generators : ");
    scanf("%d",&nbits);
    if (nbits<0)
    {
        printf("\n number of real generators entered is : %d",nbits);
        printf("\n Wrong number of generators entered, hence exiting \n");
        exit (1);
    }
    if (nbits != 0)
    {
        min_realvar = (double *)malloc(nreal*sizeof(double));
        max_realvar = (double *)malloc(nreal*sizeof(double));
        for (i=0; i<nbits; i++)
        {
            for ( hr=0; hr <24; hr++)
                {
                printf ("\n Enter the output for generator %d : ",i+1);
                scanf ("%lf",&xreal[i][hr]);
                printf ("\n Enter the cost rate for generator %d : ",i+1);
                scanf ("%lf",&gen_cost[i][hr]);
                }
            printf ("\n Enter the lower limit of real variable %d : ",i+1);
            scanf ("%lf",&min_realvar[i]);
            printf ("\n Enter the upper limit of real variable %d : ",i+1);
            scanf ("%lf",&max_realvar[i]);
            if (max_realvar[i] <= min_realvar[i])
            {
                printf("\n Wrong limits entered for the min and max bounds of generator %d, hence
exiting \n",i+1);
                exit(1);
            }
        }
        printf ("\n Enter the probability of crossover (0.6-1.0) : ");
        scanf ("%lf",&pcross_real);
```

```
        if (pcross_real<0.0 || pcross_real>1.0)
        {
            printf("\n Probability of crossover entered is : %e",pcross_real);
            printf("\n Entered value of probability of crossover of real variables is out of bounds,
hence exiting \n");
            exit (1);
        }
        printf ("\n Enter the probablity of mutation (1/nreal) : ");
        scanf ("%lf",&pmut_real);
        if (pmut_real<0.0 || pmut_real>1.0)
        {
            printf("\n Probability of mutation entered is : %e",pmut_real);
            printf("\n Entered value of probability of mutation of real variables is out of bounds,
hence exiting \n");
            exit (1);
        }
        printf ("\n Enter the value of distribution index for crossover (5-20): ");
        scanf ("%lf",&eta_c);
        if (eta_c<=0)
        {
            printf("\n The value entered is : %e",eta_c);
            printf("\n Wrong value of distribution index for crossover entered, hence exiting \n");
            exit (1);
        }
        printf ("\n Enter the value of distribution index for mutation (5-50): ");
        scanf ("%lf",&eta_m);
        if (eta_m<=0)
        {
            printf("\n The value entered is : %e",eta_m);
            printf("\n Wrong value of distribution index for mutation entered, hence exiting \n");
            exit (1);
        }
    }


    if (nbits==0)
    {
        printf("\n Number of variables is zero, hence exiting \n");
        exit(1);
    }
    printf("\n Input data successfully entered, now performing initialization \n");
    fprintf(fpt5,"\n Population size = %d",popsize);
    fprintf(fpt5,"\n Number of function evaluations = %d",neval);
    fprintf(fpt5,"\n Number of objective functions = %d",nobj);
    for (i=0; i<nobj; i++)
    {
```

```
      fprintf(fpt5,"\n Epsilon for objective %d = %e",i+1,epsilon[i]);
      fprintf(fpt5,"\n Minimum value of objective %d = %e",i+1,min_obj[i]);
   }
   fprintf(fpt5,"\n Number of constraints = %d",ncon);
   fprintf(fpt5,"\n Number of real variables = %d",nreal);
   if (nreal!=0)
   {
      for (i=0; i<nbits; i++)
      {
         fprintf(fpt5,"\n Lower limit of real variable %d = %e",i+1,min_realvar[i]);
         fprintf(fpt5,"\n Upper limit of real variable %d = %e",i+1,max_realvar[i]);
      }
      fprintf(fpt5,"\n Probability of crossover of real variable = %e",pcross_real);
      fprintf(fpt5,"\n Probability of mutation of real variable = %e",pmut_real);
      fprintf(fpt5,"\n Distribution index for crossover = %e",eta_c);
      fprintf(fpt5,"\n Distribution index for mutation = %e",eta_m);
   }
   fprintf(fpt5,"\n Number of binary variables = %d",nbin);

   fprintf(fpt5,"\n Seed for random number generator = %e",seed);
   bitlength = 0;

   fprintf(fpt1,"# of objectives = %d, # of constraints = %d, # of real_var = %d, # of bits of
bin_var = %d, constr_violation\n",nobj,ncon,nreal,bitlength);
   fprintf(fpt2,"# of objectives = %d, # of constraints = %d, # of real_var = %d, # of bits of
bin_var = %d, constr_violation\n",nobj,ncon,nreal,bitlength);
   fprintf(fpt3,"# of objectives = %d, # of constraints = %d, # of real_var = %d, # of bits of
bin_var = %d, constr_violation\n",nobj,ncon,nreal,bitlength);
   fprintf(fpt4,"# of objectives = %d, # of constraints = %d, # of real_var = %d, # of bits of
bin_var = %d, constr_violation\n",nobj,ncon,nreal,bitlength);
   nbinmut = 0;
   nrealmut = 0;
   nbincross = 0;
   nrealcross = 0;
   currenteval = 0;
   elite_size = 0;
   randomize();
   ea = (individual *)malloc(popsize*sizeof(individual));
   array = (int *)malloc(popsize*sizeof(int));
   for (i=0; i<popsize; i++)
   {
      allocate (&ea[i]);
      initialize(&ea[i]);
      decode(&ea[i]);
      eval(&ea[i]);
   }
```

```
report_pop (ea, fpt1);
elite = (ind_list *)malloc(sizeof(ind_list));
elite->ind = (individual *)malloc(sizeof(individual));
allocate (elite->ind);
elite->parent = NULL;
elite->child = NULL;
insert (elite, &ea[0]);
for (i=1; i<popsize; i++)
{
   update_elite (elite, &ea[i]);
}
child1 = (individual *)malloc(sizeof(individual));
allocate (child1);
child2 = (individual *)malloc(sizeof(individual));
allocate (child2);
cur = elite;
while (currenteval<neval)
{
   index1 = rnd(0, popsize-1);
   index2 = rnd(0, popsize-1);
   parent1 = tournament (&ea[index1], &ea[index2]);
   index = rnd(0, elite_size-1);
   cur = elite->child;
   for (i=1; i<=index; i++)
   {
      cur=cur->child;
   }
   parent2 = cur->ind;
   crossover (parent1, parent2, child1, child2);
   mutation (child1);
   decode (child1);
   eval (child1);
   update_elite (elite, child1);
   update_pop (ea, child1);
   mutation (child2);
   decode (child2);
   eval (child2);
   update_elite (elite, child2);
   update_pop (ea, child2);
   printf("\n Currenteval = %d and Elite_size = %d",currenteval,elite_size);
               /* Comment following three lines if information at all
               evaluation is not desired, it will speed up execution of the code */
      fprintf(fpt4,"# eval id = %d\n",currenteval);
               report_archive (elite, fpt4);
               fflush(fpt4);
}
```

```
printf("\n Generations finished, now reporting solutions");
report_pop (ea, fpt2);
report_archive (elite, fpt3);
if (nreal!=0)
{
   fprintf(fpt5,"\n Number of crossover of real variable = %d",nrealcross);
   fprintf(fpt5,"\n Number of mutation of real variable = %d",nrealmut);
}
if (nbin!=0)
{
   fprintf(fpt5,"\n Number of crossover of binary variable = %d",nbincross);
   fprintf(fpt5,"\n Number of mutation of binary variable = %d",nbinmut);
}
fflush(stdout);
fflush(fpt1);
fflush(fpt2);
fflush(fpt3);
fflush(fpt4);
fflush(fpt5);
fclose(fpt1);
fclose(fpt2);
fclose(fpt3);
fclose(fpt4);
fclose(fpt5);
if (nreal!=0)
{
   free (min_realvar);
   free (max_realvar);
}
if (nbin!=0)
{
   free (min_binvar);
   free (max_binvar);
   free (nbits);
}
free (epsilon);
free (min_obj);
free (array);
for (i=0; i<popsize; i++)
{
   deallocate (&ea[i]);
}
free (ea);
cur = elite->child;
while (cur!=NULL)
{
```

```
        cur = del(cur);
        cur = cur->child;
    }
    deallocate (elite->ind);
    free (elite->ind);
    free (elite);
    printf("\n Routine successfully exited \n");
    return (0);
}

/* Routine for evaluating individuals  */

# include <stdio.h>
# include <stdlib.h>
# include <math.h>

# include "global.h"
# include "rand.h"

/* Routine to evaluate objective function values and constraints for an individual */
void eval (individual *ind)
{
    int j;
    test_problem (ind->xreal,ind->gen_cost, ind->gene, ind->obj, ind->constr);
    for (j=0; j<nobj; j++)
    {
        ind->ia[j] = (int)floor( (ind->obj[j]-min_obj[j])/epsilon[j] );
    }
    if (ncon==0)
    {
        ind->constr_violation = 0.0;
    }
    else
    {
        ind->constr_violation = 0.0;
        for (j=0; j<ncon; j++)
        {
            if (ind->constr[j]<0.0)
            {
                ind->constr_violation += ind->constr[j];
            }
        }
    }
    currenteval++;
    return;
}
```

```
/* This file contains the variable and function declarations */

# ifndef _GLOBAL_H_
# define _GLOBAL_H_

# define INF 1.0e99
# define EPS 1.0e-14
# define E  2.71828182845905
# define PI 3.14159265358979

/* global variables */
typedef struct
{
   double constr_violation;
   double **xreal;
   int *gene;
   double *gen_cost;
   double *obj;
   int *ia;
   double *constr;
} individual;

typedef struct ind_lists
{
   individual *ind;
   struct ind_lists *parent;
   struct ind_lists *child;
} ind_list;

extern int nreal;
extern int nbin;
extern int nobj;
extern int ncon;
extern int popsize;
extern double pcross_real;
extern double pcross_bin;
extern double pmut_real;
extern double pmut_bin;
extern double eta_c;
extern double eta_m;
extern int neval;
extern int currenteval;
extern int nbinmut;
extern int nrealmut;
extern int nbincross;
```

```
extern int nrealcross;
extern int nbits;
extern int *array;
extern int *right_genes;
extern int *assigned;
extern double max_gen;
extern double *min_realvar;
extern double *max_realvar;
extern double *min_binvar;
extern double *max_binvar;
extern double *epsilon;
extern double *min_obj;
extern int bitlength;
extern int elite_size;
extern int site1;
/* global function declarations */
void allocate (individual *ind);
void deallocate (individual *ind);

void copy (individual *ind1, individual *ind2);

void crossover (individual *parent1, individual *parent2, individual *child1, individual *child2);
void realcross (individual *parent1, individual *parent2, individual *child1, individual *child2);
void bincross (individual *parent1, individual *parent2, individual *child1, individual *child2);

void decode (individual *ind);

int check_box_dominance (individual *a, individual *b);
int check_dominance (individual *a, individual *b);

void eval (individual *ind);

void initialize (individual *ind);

void insert (ind_list *node, individual *ind);
ind_list* del (ind_list *node);

void mutation (individual *ind);
void bin_mutate (individual *ind);
void real_mutate (individual *ind);

void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr);

void report_pop (individual *ind, FILE *fpt);
void report_archive (ind_list *elite, FILE *fpt);
```

```
individual* tournament (individual *ind1, individual *ind2);

void update_elite (ind_list *elite, individual *ind);
void update_pop (individual *ea, individual *ind);

# endif

/* Data initializtion routines */

# include <stdio.h>
# include <stdlib.h>
# include <math.h>

# include "global.h"
# include "rand.h"

/* Function to initialize an individual randomly */
void initialize (individual *ind, int site1)
{
   int i, j, k, found, found2,res;
   if (nbits !=0)
   {

   for (i=0; i < (nbits - site1); i++)
      {
      max_gen[i]= 0.0;
      for (j=0; j<nbits; j++)
         {
         if (i == 0)
            {
             if (max_gen[i] < gen_cost[j])
                max_gen[i]= gen_cost[j];
            }
         else
            {
            if ((max_gen[i] < gen_cost[j]) && (max_gen[i] < max_gen[i-1]))
              max_gen[i]= gen_cost[j];
            }
         }
    /*fetch a single random integer at a time to select each gene to the right of the crossover point
and is one of the expensive generators */
    k =0;
    j= site1;
    right_genes[0] =10000;

        while (k < nbits -site1)
```

```
        {
        res = rnd[0,nbits];

           if ( gen_cost[res] >= max_gen[nbits -site1-1])
             {
             found =0;
              for (i=0; i < k ; i++)
                 if (res == right_genes[i])
                   found=1;
                 if ( found == 0)
                   {
                   ind->gene[j] = res;
                   for ( hr=0; hr <24; hr++)
                      ind->xreal[j][hr]= rndreal (min_realvar[res], max_realvar[res]);
                   right_genes[k] = res;
                   j++;
                   k++;
                   }
             }
         }
/* next assign all remaining generators not assigned to the appropreate gene */
jump=k;
j=0;
assigned[0]=10000;
while ( j < site1)
    {
    res = rnd[0,bits];
    found =0;
    found2=0;
    for (k=0 ; k < jump; k++)
       if( res == right_genes[k])
         found =1;

    for ( i=0; i < j; i++)
       if( res == assigned[i] )
         found2=1;
            if ( found1 ==0 && found2 ==0)
              {
              ind->gene[j] = res;
              for ( hr= 0 ; hr < 24; hr++)
                 {
                 ind->xreal[j][hr]= rndreal (min_realvar[res], max_realvar[res]);

                 }
              assigned [j] =res;
              j++;
```

```
                    }


            }
        }
        return;
}

/* A custom doubly linked list implemenation */

# include <stdio.h>
# include <stdlib.h>
# include <math.h>

# include "global.h"
# include "rand.h"

/* Routine to insert an element after the location specified by node NODE */
void insert (ind_list *node, individual *ind)
{
    ind_list *temp;
    if (node==NULL)
    {
        printf("\n Error!! asked to enter after a NULL pointer, hence exiting \n");
        exit(1);
    }
    temp = (ind_list *)malloc(sizeof(ind_list));
    temp->ind = (individual *)malloc(sizeof(individual));
    allocate (temp->ind);
    copy (ind, temp->ind);
    temp->child = node->child;
    temp->parent = node;
    if (node->child != NULL)
    {
        node->child->parent = temp;
    }
    node->child = temp;
    elite_size++;
    return;
}

/* Delete the element specified by node NODE */
ind_list* del (ind_list *node)
{
    ind_list *temp;
    if (node==NULL)
```

```
    {
       printf("\n Error!! asked to delete a NULL pointer, hence exiting \n");
       exit(1);
    }
    temp = node->parent;
    temp->child = node->child;
    if (temp->child!=NULL)
    {
       temp->child->parent = temp;
    }
    deallocate(node->ind);
    free (node->ind);
    free (node);
    elite_size--;
    return (temp);
}

/* Mutation routines */

# include <stdio.h>
# include <stdlib.h>
# include <math.h>

# include "global.h"
# include "rand.h"

/* Function to perform mutation of an individual */
void mutation (individual *ind)
{
    if (nreal!=0)
    {
       real_mutate(ind);
    }
    if (nbits!=0)
    {
       bin_mutate(ind);
    }
    return;
}

/* Routine for binary mutation of an individual */
void bin_mutate (individual *ind)
{
    int j, k, found,found2,temp;
    double prob, temp_real[24],new_real[24];
```

```
  for( hr=0; hr <24; hr++)
    {
    temp_real[24]=0.0;
    new_real[24]=0.0;
    }

        prob = randomperc();
        if (prob <=pmut_bin)
        {
        found =0;
        res = rnd[0,site1];
        while (found !=1 )
           {
           res1 = rnd[0,site1];
           if ( res !=res1)
             {

             for( hr=0; hr <24; hr++)
               temp_real[24] = ind->xreal[res][hr]+ind->xreal[res1][hr];
             temp = ind->gene[res];
             ind-> gene[res] = ind->gene[res1];
             ind-> gene[res1] = temp;
             for( hr=0; hr <24; hr++)
               {
               found2=0;
               while( found2 !=1)
                   {
                   ind->xreal[res][hr] =rndreal (min_realvar[res], max_realvar[res]);
                   new_real[24] = temp_real[24] - ind->xreal[res][hr];

                   if ( new_real[24] > min_realvar[res1])&&( new_real[24] < max_realvar[res1]))
                     {
                     ind->xreal[res1][hr] = new_real[24];
                     found2=1;
                     }
                   }
                }
             found=1;
             }
           }
        }

  return;
}
```

```
/* Routine for real polynomial mutation of an individual */
void real_mutate (individual *ind)
{
   int j;
   double rnd, delta1, delta2, mut_pow, deltaq;
   double y, yl, yu, val, xy;
   for (j=0; j<nreal; j++)
   {
                  if (randomperc() <= pmut_real)
      {
         y = ind->xreal[j];
         yl = min_realvar[j];
         yu = max_realvar[j];
         delta1 = (y-yl)/(yu-yl);
         delta2 = (yu-y)/(yu-yl);
         rnd = randomperc();
         mut_pow = 1.0/(eta_m+1.0);
         if (rnd <= 0.5)
         {
            xy = 1.0-delta1;
            val = 2.0*rnd+(1.0-2.0*rnd)*(pow(xy,(eta_m+1.0)));
            deltaq =  pow(val,mut_pow) - 1.0;
         }
         else
         {
            xy = 1.0-delta2;
            val = 2.0*(1.0-rnd)+2.0*(rnd-0.5)*(pow(xy,(eta_m+1.0)));
            deltaq = 1.0 - (pow(val,mut_pow));
         }
         y = y + deltaq*(yu-yl);
         if (y<yl)         y = yl;
                           if (y>yu)         y = yu;
                           ind->xreal[j] = y;
                           nrealmut+=1;
                  }
   }
         return;
}

/* Test problem definitions */

# include <stdio.h>
# include <stdlib.h>
# include <math.h>

# include "global.h"
```

```
# include "rand.h"

/* # define sch1 */
/* # define sch2 */
/* # define fon */
/* # define kur */
/* # define pol */
/* # define vnt */
/* # define zdt1*/
/* # define zdt2 */
/* # define zdt3 */
/* # define zdt4 */
/* # define zdt5 */
/* # define zdt6 */
/* # define bnh */
/* # define osy */
/* # define srn */
/* # define tnk */
/* # define ctp1 */
/* # define ctp2 */
/* # define ctp3 */
/* # define ctp4 */
/* # define ctp5 */
/* # define ctp6 */
/* # define ctp7 */
/* # define ctp8 */
#define generator_matchup
/*  Test problem SCH1
   # of real variables = 1
   # of bin variables = 0
   # of objectives = 2
   # of constraints = 0
   */

#ifdef sch1
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   obj[0] = pow(xreal[0],2.0);
   obj[1] = pow((xreal[0]-2.0),2.0);
   return;
}
#endif

/*  Test problem SCH2
   # of real variables = 1
   # of bin variables = 0
```

```
   # of objectives = 2
   # of constraints = 0
   */

#ifdef sch2
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   if (xreal[0]<=1.0)
   {
      obj[0] = -xreal[0];
      obj[1] = pow((xreal[0]-5.0),2.0);
      return;
   }
   if (xreal[0]<=3.0)
   {
      obj[0] = xreal[0]-2.0;
      obj[1] = pow((xreal[0]-5.0),2.0);
      return;
   }
   if (xreal[0]<=4.0)
   {
      obj[0] = 4.0-xreal[0];
      obj[1] = pow((xreal[0]-5.0),2.0);
      return;
   }
   obj[0] = xreal[0]-4.0;
   obj[1] = pow((xreal[0]-5.0),2.0);
   return;
}
#endif

/*  Test problem FON
   # of real variables = n
   # of bin variables = 0
   # of objectives = 2
   # of constraints = 0
   */

#ifdef fon
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   double s1, s2;
   int i;
   s1 = s2 = 0.0;
   for (i=0; i<nreal; i++)
   {
```

```
        s1 += pow((xreal[i]-(1.0/sqrt((double)nreal))),2.0);
        s2 += pow((xreal[i]+(1.0/sqrt((double)nreal))),2.0);
    }
    obj[0] = 1.0 - exp(-s1);
    obj[1] = 1.0 - exp(-s2);
    return;
}
#endif

/*  Test problem KUR
    # of real variables = 3
    # of bin variables = 0
    # of objectives = 2
    # of constraints = 0
    */

#ifdef kur
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
    int i;
    double res1, res2;
    res1 = -0.2*sqrt((xreal[0]*xreal[0]) + (xreal[1]*xreal[1]));
    res2 = -0.2*sqrt((xreal[1]*xreal[1]) + (xreal[2]*xreal[2]));
    obj[0] = -10.0*( exp(res1) + exp(res2));
    obj[1] = 0.0;
    for (i=0; i<3; i++)
    {
        obj[1] += pow(fabs(xreal[i]),0.8) + 5.0*sin(pow(xreal[i],3.0));
    }
    return;
}
#endif

/*  Test problem POL
    # of real variables = 2
    # of bin variables = 0
    # of objectives = 2
    # of constraints = 0
    */

#ifdef pol
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
    double a1, a2, b1, b2;
    a1 = 0.5*sin(1.0) - 2.0*cos(1.0) + sin(2.0) - 1.5*cos(2.0);
    a2 = 1.5*sin(1.0) - cos(1.0) + 2.0*sin(2.0) - 0.5*cos(2.0);
```

```
   b1 = 0.5*sin(xreal[0]) - 2.0*cos(xreal[0]) + sin(xreal[1]) - 1.5*cos(xreal[1]);
   b2 = 1.5*sin(xreal[0]) - cos(xreal[0]) + 2.0*sin(xreal[1]) - 0.5*cos(xreal[1]);
   obj[0] = 1.0 + pow((a1-b1),2.0) + pow((a2-b2),2.0);
   obj[1] = pow((xreal[0]+3.0),2.0) + pow((xreal[1]+1.0),2.0);
   return;
}
#endif

/*  Test problem VNT
   # of real variables = 2
   # of bin variables = 0
   # of objectives = 3
   # of constraints = 0
   */

#ifdef vnt
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   obj[0] = 0.5*(xreal[0]*xreal[0] + xreal[1]*xreal[1]) + sin(xreal[0]*xreal[0] +
xreal[1]*xreal[1]);
   obj[1] = (pow((3.0*xreal[0] - 2.0*xreal[1] + 4.0),2.0))/8.0 + (pow((xreal[0]-
xreal[1]+1.0),2.0))/27.0 + 15.0;
   obj[2] = 1.0/(xreal[0]*xreal[0] + xreal[1]*xreal[1] + 1.0) - 1.1*exp(-(xreal[0]*xreal[0] +
xreal[1]*xreal[1]));
   return;
}
#endif

/*  Test problem ZDT1
   # of real variables = 30
   # of bin variables = 0
   # of objectives = 2
   # of constraints = 0
   */

#ifdef zdt1
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   double f1, f2, g, h;
   int i;
   f1 = xreal[0];
   g = 0.0;
   for (i=1; i<30; i++)
   {
      g += xreal[i];
   }
```

```
    g = 9.0*g/29.0;
    g += 1.0;
    h = 1.0 - sqrt(f1/g);
    f2 = g*h;
    obj[0] = f1;
    obj[1] = f2;
    return;
}
#endif

/*  Test problem ZDT2
    # of real variables = 30
    # of bin variables = 0
    # of objectives = 2
    # of constraints = 0
    */

#ifdef zdt2
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
    double f1, f2, g, h;
    int i;
    f1 = xreal[0];
    g = 0.0;
    for (i=1; i<30; i++)
    {
        g += xreal[i];
    }
    g = 9.0*g/29.0;
    g += 1.0;
    h = 1.0 - pow((f1/g),2.0);
    f2 = g*h;
    obj[0] = f1;
    obj[1] = f2;
    return;
}
#endif

/*  Test problem ZDT3
    # of real variables = 30
    # of bin variables = 0
    # of objectives = 2
    # of constraints = 0
    */

#ifdef zdt3
```

```
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   double f1, f2, g, h;
   int i;
   f1 = xreal[0];
   g = 0.0;
   for (i=1; i<30; i++)
   {
      g += xreal[i];
   }
   g = 9.0*g/29.0;
   g += 1.0;
   h = 1.0 - sqrt(f1/g) - (f1/g)*sin(10.0*PI*f1);
   f2 = g*h;
   obj[0] = f1;
   obj[1] = f2;
   return;
}
#endif

/*  Test problem ZDT4
   # of real variables = 10
   # of bin variables = 0
   # of objectives = 2
   # of constraints = 0
   */

#ifdef zdt4
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   double f1, f2, g, h;
   int i;
   f1 = xreal[0];
   g = 0.0;
   for (i=1; i<10; i++)
   {
      g += xreal[i]*xreal[i] - 10.0*cos(4.0*PI*xreal[i]);
   }
   g += 91.0;
   h = 1.0 - sqrt(f1/g);
   f2 = g*h;
   obj[0] = f1;
   obj[1] = f2;
   return;
}
#endif
```

```
/*  Test problem ZDT5
    # of real variables = 0
    # of bin variables = 11
    # of bits for binvar1 = 30
    # of bits for binvar2-11 = 5
    # of objectives = 2
    # of constraints = 0
    */

#ifdef zdt5
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
    int i, j;
    int u[11];
    int v[11];
    double f1, f2, g, h;
    for (i=0; i<11; i++)
    {
        u[i] = 0;
    }
    for (j=0; j<30; j++)
    {
        if (gene[0][j] == 1)
        {
            u[0]++;
        }
    }
    for (i=1; i<11; i++)
    {
        for (j=0; j<4; j++)
        {
            if (gene[i][j] == 1)
            {
                u[i]++;
            }
        }
    }
    f1 = 1.0 + u[0];
    for (i=1; i<11; i++)
    {
        if (u[i] < 5)
        {
            v[i] = 2 + u[i];
        }
        else
```

```
        {
            v[i] = 1;
        }
    }
    g = 0;
    for (i=1; i<11; i++)
    {
        g += v[i];
    }
    h = 1.0/f1;
    f2 = g*h;
    obj[0] = f1;
    obj[1] = f2;
    return;
}
#endif

/*  Test problem ZDT6
    # of real variables = 10
    # of bin variables = 0
    # of objectives = 2
    # of constraints = 0
    */

#ifdef zdt6
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
    double f1, f2, g, h;
    int i;
    f1 = 1.0 - (exp(-4.0*xreal[0]))*pow((sin(4.0*PI*xreal[0])),6.0);
    g = 0.0;
    for (i=1; i<10; i++)
    {
        g += xreal[i];
    }
    g = g/9.0;
    g = pow(g,0.25);
    g = 1.0 + 9.0*g;
    h = 1.0 - pow((f1/g),2.0);
    f2 = g*h;
    obj[0] = f1;
    obj[1] = f2;
    return;
}
#endif
```

```
/*  Test problem BNH
    # of real variables = 2
    # of bin variables = 0
    # of objectives = 2
    # of constraints = 2
    */

#ifdef bnh
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
    obj[0] = 4.0*(xreal[0]*xreal[0] + xreal[1]*xreal[1]);
    obj[1] = pow((xreal[0]-5.0),2.0) + pow((xreal[1]-5.0),2.0);
    constr[0] = 1.0 - (pow((xreal[0]-5.0),2.0) + xreal[1]*xreal[1])/25.0;
    constr[1] = (pow((xreal[0]-8.0),2.0) + pow((xreal[1]+3.0),2.0))/7.7 - 1.0;
    return;
}
#endif

/*  Test problem OSY
    # of real variables = 6
    # of bin variables = 0
    # of objectives = 2
    # of constraints = 6
    */

#ifdef osy
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
    obj[0] = -(25.0*pow((xreal[0]-2.0),2.0) + pow((xreal[1]-2.0),2.0) + pow((xreal[2]-1.0),2.0) +
pow((xreal[3]-4.0),2.0) + pow((xreal[4]-1.0),2.0));
    obj[1] = xreal[0]*xreal[0] +  xreal[1]*xreal[1] + xreal[2]*xreal[2] + xreal[3]*xreal[3] +
xreal[4]*xreal[4] + xreal[5]*xreal[5];
    constr[0] = (xreal[0]+xreal[1])/2.0 - 1.0;
    constr[1] = 1.0 - (xreal[0]+xreal[1])/6.0;
    constr[2] = 1.0 - xreal[1]/2.0 + xreal[0]/2.0;
    constr[3] = 1.0 - xreal[0]/2.0 + 3.0*xreal[1]/2.0;
    constr[4] = 1.0 - (pow((xreal[2]-3.0),2.0))/4.0 - xreal[3]/4.0;
    constr[5] = (pow((xreal[4]-3.0),2.0))/4.0 + xreal[5]/4.0 - 1.0;
    return;
}
#endif

/*  Test problem SRN
    # of real variables = 2
    # of bin variables = 0
    # of objectives = 2
```

```
   # of constraints = 2
   */

#ifdef srn
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   obj[0] = 2.0 + pow((xreal[0]-2.0),2.0) + pow((xreal[1]-1.0),2.0);
   obj[1] = 9.0*xreal[0] - pow((xreal[1]-1.0),2.0);
   constr[0] = 1.0 - (pow(xreal[0],2.0) + pow(xreal[1],2.0))/225.0;
   constr[1] = 3.0*xreal[1]/10.0 - xreal[0]/10.0 - 1.0;
   return;
}
#endif

/*  Test problem TNK
   # of real variables = 2
   # of bin variables = 0
   # of objectives = 2
   # of constraints = 2
   */

#ifdef tnk
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   obj[0] = xreal[0];
   obj[1] = xreal[1];
   if (xreal[1] == 0.0)
   {
      constr[0] = -1.0;
   }
   else
   {
      constr[0] = xreal[0]*xreal[0] + xreal[1]*xreal[1] - 0.1*cos(16.0*atan(xreal[0]/xreal[1])) -
1.0;
   }
   constr[1] = 1.0 - 2.0*pow((xreal[0]-0.5),2.0) + 2.0*pow((xreal[1]-0.5),2.0);
   return;
}
#endif

/*  Test problem CTP1
   # of real variables = 2
   # of bin variables = 0
   # of objectives = 2
   # of constraints = 2
   */
```

```
#ifdef ctp1
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   double g;
   g = 1.0 + xreal[1];
   obj[0] = xreal[0];
   obj[1] = g*exp(-obj[0]/g);
   constr[0] = obj[1]/(0.858*exp(-0.541*obj[0]))-1.0;
   constr[1] = obj[1]/(0.728*exp(-0.295*obj[0]))-1.0;
   return;
}
#endif

/*  Test problem CTP2
   # of real variables = 2
   # of bin variables = 0
   # of objectives = 2
   # of constraints = 1
   */

#ifdef ctp2
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   double g;
   double theta, a, b, c, d, e;
   double exp1, exp2;
   theta = -0.2*PI;
   a = 0.2;
   b = 10.0;
   c = 1.0;
   d = 6.0;
   e = 1.0;
   g = 1.0 + xreal[1];
   obj[0] = xreal[0];
   obj[1] = g*(1.0  - sqrt(obj[0]/g));
   exp1 = (obj[1]-e)*cos(theta) - obj[0]*sin(theta);
   exp2 = (obj[1]-e)*sin(theta) + obj[0]*cos(theta);
   exp2 = b*PI*pow(exp2,c);
   exp2 = fabs(sin(exp2));
   exp2 = a*pow(exp2,d);
   constr[0] = exp1/exp2 - 1.0;
   return;
}
#endif
```

```
/*  Test problem CTP3
   # of real variables = 2
   # of bin variables = 0
   # of objectives = 2
   # of constraints = 1
   */

#ifdef ctp3
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   double g;
   double theta, a, b, c, d, e;
   double exp1, exp2;
   theta = -0.2*PI;
   a = 0.1;
   b = 10.0;
   c = 1.0;
   d = 0.5;
   e = 1.0;
   g = 1.0 + xreal[1];
   obj[0] = xreal[0];
   obj[1] = g*(1.0  - sqrt(obj[0]/g));
   exp1 = (obj[1]-e)*cos(theta) - obj[0]*sin(theta);
   exp2 = (obj[1]-e)*sin(theta) + obj[0]*cos(theta);
   exp2 = b*PI*pow(exp2,c);
   exp2 = fabs(sin(exp2));
   exp2 = a*pow(exp2,d);
   constr[0] = exp1/exp2 - 1.0;
   return;
}
#endif

/*  Test problem CTP4
   # of real variables = 2
   # of bin variables = 0
   # of objectives = 2
   # of constraints = 1
   */

#ifdef ctp4
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   double g;
   double theta, a, b, c, d, e;
   double exp1, exp2;
   theta = -0.2*PI;
```

```
    a = 0.75;
    b = 10.0;
    c = 1.0;
    d = 0.5;
    e = 1.0;
    g = 1.0 + xreal[1];
    obj[0] = xreal[0];
    obj[1] = g*(1.0  - sqrt(obj[0]/g));
    exp1 = (obj[1]-e)*cos(theta) - obj[0]*sin(theta);
    exp2 = (obj[1]-e)*sin(theta) + obj[0]*cos(theta);
    exp2 = b*PI*pow(exp2,c);
    exp2 = fabs(sin(exp2));
    exp2 = a*pow(exp2,d);
    constr[0] = exp1/exp2 - 1.0;
    return;
}
#endif

/*  Test problem CTP5
    # of real variables = 2
    # of bin variables = 0
    # of objectives = 2
    # of constraints = 1
    */

#ifdef ctp5
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
    double g;
    double theta, a, b, c, d, e;
    double exp1, exp2;
    theta = -0.2*PI;
    a = 0.1;
    b = 10.0;
    c = 2.0;
    d = 0.5;
    e = 1.0;
    g = 1.0 + xreal[1];
    obj[0] = xreal[0];
    obj[1] = g*(1.0  - sqrt(obj[0]/g));
    exp1 = (obj[1]-e)*cos(theta) - obj[0]*sin(theta);
    exp2 = (obj[1]-e)*sin(theta) + obj[0]*cos(theta);
    exp2 = b*PI*pow(exp2,c);
    exp2 = fabs(sin(exp2));
    exp2 = a*pow(exp2,d);
    constr[0] = exp1/exp2 - 1.0;
```

```
   return;
}
#endif

/*  Test problem CTP6
   # of real variables = 2
   # of bin variables = 0
   # of objectives = 2
   # of constraints = 1
   */

#ifdef ctp6
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
   double g;
   double theta, a, b, c, d, e;
   double exp1, exp2;
   theta = 0.1*PI;
   a = 40.0;
   b = 0.5;
   c = 1.0;
   d = 2.0;
   e = -2.0;
   g = 1.0 + xreal[1];
   obj[0] = xreal[0];
   obj[1] = g*(1.0  - sqrt(obj[0]/g));
   exp1 = (obj[1]-e)*cos(theta) - obj[0]*sin(theta);
   exp2 = (obj[1]-e)*sin(theta) + obj[0]*cos(theta);
   exp2 = b*PI*pow(exp2,c);
   exp2 = fabs(sin(exp2));
   exp2 = a*pow(exp2,d);
   constr[0] = exp1/exp2 - 1.0;
   return;
}
#endif

/*  Test problem CTP7
   # of real variables = 2
   # of bin variables = 0
   # of objectives = 2
   # of constraints = 1
   */

#ifdef ctp7
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
```

```
    double g;
    double theta, a, b, c, d, e;
    double exp1, exp2;
    theta = -0.05*PI;
    a = 40.0;
    b = 5.0;
    c = 1.0;
    d = 6.0;
    e = 0.0;
    g = 1.0 + xreal[1];
    obj[0] = xreal[0];
    obj[1] = g*(1.0  - sqrt(obj[0]/g));
    exp1 = (obj[1]-e)*cos(theta) - obj[0]*sin(theta);
    exp2 = (obj[1]-e)*sin(theta) + obj[0]*cos(theta);
    exp2 = b*PI*pow(exp2,c);
    exp2 = fabs(sin(exp2));
    exp2 = a*pow(exp2,d);
    constr[0] = exp1/exp2 - 1.0;
    return;
}
#endif

/*  Test problem CTP8
    # of real variables = 2
    # of bin variables = 0
    # of objectives = 2
    # of constraints = 2
    */

#ifdef ctp8
void test_problem (double *xreal, double *xbin, int **gene, double *obj, double *constr)
{
    double g;
    double theta, a, b, c, d, e;
    double exp1, exp2;
    g = 1.0 + xreal[1];
    obj[0] = xreal[0];
    obj[1] = g*(1.0  - sqrt(obj[0]/g));
    theta = 0.1*PI;
    a = 40.0;
    b = 0.5;
    c = 1.0;
    d = 2.0;
    e = -2.0;
    exp1 = (obj[1]-e)*cos(theta) - obj[0]*sin(theta);
    exp2 = (obj[1]-e)*sin(theta) + obj[0]*cos(theta);
```

```
    exp2 = b*PI*pow(exp2,c);
    exp2 = fabs(sin(exp2));
    exp2 = a*pow(exp2,d);
    constr[0] = exp1/exp2 - 1.0;
    theta = -0.05*PI;
    a = 40.0;
    b = 2.0;
    c = 1.0;
    d = 6.0;
    e = 0.0;
    exp1 = (obj[1]-e)*cos(theta) - obj[0]*sin(theta);
    exp2 = (obj[1]-e)*sin(theta) + obj[0]*cos(theta);
    exp2 = b*PI*pow(exp2,c);
    exp2 = fabs(sin(exp2));
    exp2 = a*pow(exp2,d);
    constr[1] = exp1/exp2 - 1.0;
    return;
}
#endif

#ifdef generator_matchup
void test_problem (double **xreal,double *gen_cost, double *load_cost, double **load, double
*max_realvar, int num_loads, int nreal, int *group, int group_max, double *obj, double *constr)
{
  int i,j,k,l, hrs=24;
  obj[0] =0.0;
  obj[1] = 0.0;
double sum =0.0, sum1=0.0;
/* Objective function definition */

    for (i = 0; i < hrs ; i++)
        {
        for ( j = 0; j < nreal ; j++)
            {
            obj[0] += (gen_cost[j]*xreal[j][i];
            }
        for ( k= 0; k < num_loads ; k++)
            {
            obj[0] -= load_cost[k]*load[k][i];
            }
        }
    for (i = 0; i < hrs ; i++)
        {
        for ( j = 0; j < nreal ; j++)
            {
            for( l = 0; l < group_max ; l++)
```

```
            {
            if ( j == group[l])
              sum1 += max_realvar[j] - xreal[j][i];
            }
          sum += max_realvar[j] -x real[j][i];
          }
        for ( j = 0; j < nreal ; j++)
          {
          for( l = 0; l < group_max ; l++)
            {
            if ( j != group[l])
              obj[1] += ((max_realvar[j]- xreal[j][i])/sum)^2;
            }
          }
         obj[1] += (sum1/sum)^2;
       }
      obj[1] = obj[1]/24;
/* constraints definition */
/* Economic Minimum and Maximum Operating Cosntaraint *

   for (i = 0; i < hrs ; i++)
       {
       for ( j = 0; j < nreal ; j++)
          {
          constr[j]=max_realvar[j] -(xreal[j][i]+ Spin[j];
          }
       }
   for (i = 0; i < hrs ; i++)
       {
       for ( j = 0; j < nreal ; j++)
          {
          constr[j+nreal]=max_realvar[j] -(xreal[j][i]+ SOper[j];
          }
       }
   for (i = 0; i < hrs ; i++)
       {
       for ( j = 0; j < nreal ; j++)
          {
          constr[j+2*nreal]=(xreal[j][i] -min_realvar[j];
          }
       }


   return;
}
#endif
```

```
/* Definition of random number generation routines */

# include <stdio.h>
# include <stdlib.h>
# include <math.h>

# include "global.h"
# include "rand.h"

double seed;
double oldrand[55];
int jrand;

/* Get seed number for random and start it up */
void randomize()
{
   int j1;
   for(j1=0; j1<=54; j1++)
   {
      oldrand[j1] = 0.0;
   }
   jrand=0;
   warmup_random (seed);
   return;
}

/* Get randomize off and running */
void warmup_random (double seed)
{
   int j1, ii;
   double new_random, prev_random;
   oldrand[54] = seed;
   new_random = 0.000000001;
   prev_random = seed;
   for(j1=1; j1<=54; j1++)
   {
      ii = (21*j1)%54;
      oldrand[ii] = new_random;
      new_random = prev_random-new_random;
      if(new_random<0.0)
      {
         new_random += 1.0;
      }
      prev_random = oldrand[ii];
   }
```

```
   advance_random ();
   advance_random ();
   advance_random ();
   jrand = 0;
   return;
}

/* Create next batch of 55 random numbers */
void advance_random ()
{
   int j1;
   double new_random;
   for(j1=0; j1<24; j1++)
   {
      new_random = oldrand[j1]-oldrand[j1+31];
      if(new_random<0.0)
      {
         new_random = new_random+1.0;
      }
      oldrand[j1] = new_random;
   }
   for(j1=24; j1<55; j1++)
   {
      new_random = oldrand[j1]-oldrand[j1-24];
      if(new_random<0.0)
      {
         new_random = new_random+1.0;
      }
      oldrand[j1] = new_random;
   }
}

/* Fetch a single random number between 0.0 and 1.0 */
double randomperc()
{
   jrand++;
   if(jrand>=55)
   {
      jrand = 1;
      advance_random();
   }
   return((double)oldrand[jrand]);
}

/* Fetch a single random integer between low and high including the bounds */
int rnd (int low, int high)
```

```
{
   int res;
   if (low >= high)
   {
      res = low;
   }
   else
   {
      res = low + (randomperc()*(high-low+1));
      if (res > high)
      {
         res = high;
      }
   }
   return (res);
}

/* Fetch a single random real number between low and high including the bounds */
double rndreal (double low, double high)
{
   return (low + (high-low)*randomperc());
}

/* Declaration for random number related variables and routines */

# ifndef _RAND_H_
# define _RAND_H_

/* Variable declarations for the random number generator */
extern double seed;
extern double oldrand[55];
extern int jrand;

/* Function declarations for the random number generator */
void randomize(void);
void warmup_random (double seed);
void advance_random (void);
double randomperc(void);
int rnd (int low, int high);
double rndreal (double low, double high);

# endif

/* Routines for storing population data into files */

# include <stdio.h>
```

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

# include "global.h"
# include "rand.h"

/* Function to print the information of a population in a file */
void report_pop (individual *ind, FILE *fpt)
{
   int i, j, k;
   for (i=0; i<popsize; i++)
   {
      for (j=0; j<nobj; j++)
      {
         fprintf(fpt,"%e\t",ind[i].obj[j]);
      }
      if (ncon!=0)
      {
         for (j=0; j<ncon; j++)
         {
            fprintf(fpt,"%e\t",ind[i].constr[j]);
         }
      }
      if (nreal!=0)
      {
         for (j=0; j<nbits; j++)
         {
            for ( hr <0; hr <24; hr++)
            fprintf(fpt,"%e\t",cur->ind->xreal[j][hr]);
            fprintf(fpt,"\n");
         }
      }
      if (nbits!=0)
      {

            for (k=0; k<nbits; k++)
            {
               fprintf(fpt,"%d\t",ind[i].gene[k]);
            }

      }
      fprintf(fpt,"%e\n",ind[i].constr_violation);
   }
   return;
}
```

```
/* Function to print the information of feasible and non-dominated population in a file */
void report_archive (ind_list *elite, FILE *fpt)
{
   int j, k;
   ind_list *cur;
   cur = elite->child;
   while (cur!=NULL)
   {
      for (j=0; j<nobj; j++)
      {
         fprintf(fpt,"%e\t",cur->ind->obj[j]);
      }
      if (ncon!=0)
      {
         for (j=0; j<ncon; j++)
         {
            fprintf(fpt,"%e\t",cur->ind->constr[j]);
         }
      }
      if (nbits!=0)
      {
         for (j=0; j<nbits; j++)
         {
          for ( hr <0; hr <24; hr++)
            fprintf(fpt,"%e\t",cur->ind->xreal[j][hr]);
            fprintf(fpt,"\n");
         }
      }
      if (nbits!=0)
      {

            for (k=0; k<nbits; k++)
            {
               fprintf(fpt,"%d\t",cur->ind->gene[k]);
            }

      }
      fprintf(fpt,"%e\n",cur->ind->constr_violation);
      cur = cur->child;
   }
   return;
}

/* Tournamenet Selections routine */

# include <stdio.h>
```

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

# include "global.h"
# include "rand.h"

/* Routine for binary neigborhood */
individual* tournament (individual *ind1, individual *ind2)
{
   int flag;
   flag = check_dominance (ind1, ind2);
   if (flag==1)
   {
      return (ind1);
   }
   if (flag==-1)
   {
      return (ind2);
   }
   if ((randomperc()) <= 0.5)
   {
      return(ind1);
   }
   else
   {
      return(ind2);
   }
}

/* Routines for updating elite and EA populations */

# include <stdio.h>
# include <stdlib.h>
# include <math.h>

# include "global.h"
# include "rand.h"

/* Routine to update archive */
void update_elite (ind_list *elite, individual *ind)
{
   int i, end, flag;
   double d1, d2;
   ind_list *temp;
   temp = elite->child;
   end = 0;
```

```
do
{
   flag = check_box_dominance (ind, temp->ind);
   switch (flag)
   {
   case 1: /* ind dominates temp->ind */
      {
         temp = del (temp);
         temp = temp->child;
         break;
      }
   case 2: /* temp->ind dominates ind */
      {
         return;
      }
   case 3: /* both are non-dominated and are in different boxes */
      {
         temp = temp->child;
         break;
      }
   case 4: /* both are non-dominated and are in same hyper-box */
      {
         end = 1;
         break;
      }
   }
}
while (end!=1 && temp!=NULL);
if (end==0)
{
   insert(elite, ind);
}
else
{
   if (flag==4)       /* in same hyperbox */
   {
      flag = check_dominance (ind, temp->ind);
      switch (flag)
      {
      case 1:
         {
            temp = del(temp);
            insert (elite, ind);
            break;
         }
      case -1:
```

```
            {
               return;
            }
         case 0:
            {
               d1 = 0.0;
               d2 = 0.0;
               for (i=0; i<nobj; i++)
               {
                  d1 += pow(((ind->obj[i]-ind->ia[i])/epsilon[i]),2.0);
                  d2 += pow(((temp->ind->obj[i]-temp->ind->ia[i])/epsilon[i]),2.0);
               }
               if (d1<=d2)
               {
                  temp = del(temp);
                  insert(elite,ind);
               }
               break;
            }
         }
      }
   }
   return;
}

/* Routine to update population */
void update_pop (individual *ea, individual *ind)
{
        int size;
        int i;
        int flag;
        size = 0;
        for (i=0; i<popsize; i++)
        {
                flag = check_dominance (ind, &ea[i]);
                switch (flag)
                {
                        case 1:
                                copy (ind, &ea[i]);
                                return;
                        case -1:
                                return;
                        case 0:
                                array[size++] = i;
                                break;
                }
```

```
        }
        if (size>0)
        {
                i = rnd(0,size-1);
                copy (ind, &ea[array[i]]);
        }
        return;
}
```

Appendix B – System Parameters

Buying Offers from Loads

The intercept of the curve with the y-axis is presented as parameter "a" while the slope is represented as parameter "b" in the table. The load characteristics for the 3 power systems considered are presented in the following Tables. The units of the x-axis for these load characteristics are in MW while the units of the y-axis will be in $/MW.

Test Case 1 – 5-Generator, 3-Load, 8-Bus Power System

Load #	a	b
1	5	0.1
2	12	0.1
3	-2.5	0.05

Test Case 2 – 10-Generator, 6-Load, 10-Bus Power System

Load #	a	b
1	70	1.0
2	-20	1.0
3	0	1.0
4	70	0.5
5	-40	1.5
6	20	2.0

Test Case 3 – 50-Generator, 20-Load, 27-Bus Power System

Load #	a	b
1	70	1.0
2	-20	1.0
3	0	1.0
4	70	0.5
5	-40	1.5
6	20	2.0
7	5	0.1
8	12	0.1
9	-2.5	0.05
10	70	0.5
11	-40	1.5
12	20	2.0
13	5	0.1
14	5	0.1
15	12	0.1
16	-2.5	0.05
17	70	0.5
18	-40	1.5
19	20	2.0
20	5	0.1

Constraints

Test Case 1 – 5-Generator, 3-Load, 8-Bus Power System

a) Branch Capacity Limit Constraints

$$|BFlow_{12,t}| \leq 45$$
$$|BFlow_{13,t}| \leq 20$$
$$|BFlow_{34,t}| \leq 40$$
$$|BFlow_{34,t}| \leq 30$$
$$|BFlow_{45,t}| \leq 40$$
$$|BFlow_{56,t}| \leq 50$$
$$|BFlow_{57,t}| \leq 35$$
$$|BFlow_{67,t}| \leq 45$$
$$|BFlow_{78,t}| \leq 50$$
$$|BFlow_{28,t}| \leq 30$$

b) Generator Ramp Rate Constraints

$$S_1(t\text{-}1) - S_1(t) \leq 30$$

$$S_2(t\text{-}1) - S_2(t) \leq 50$$

$$S_3(t\text{-}1) - S_3(t) \leq 60$$

$$S_4(t\text{-}1) - S_4(t) \leq 40$$

$$S_5(t\text{-}1) - S_5(t) \leq 60$$

c) Economic Maximum and Minimum Operating Constraints

$$S_1(t) + S^{spin}{}_1(t) \leq 63$$

$$S_2(t) + S^{spin}{}_2(t) \leq 90$$

$$S_3(t) + S^{spin}{}_3(t) \leq 90$$

$$S_4(t) + S^{spin}{}_4(t) \leq 54$$

$$S_5(t) + S^{spin}{}_5(t) \leq 90$$

$$S^{spin}{}_1(t) + S^{spin}{}_2(t) + S^{spin}{}_3(t) + S^{spin}{}_4(t) + S^{spin}{}_5(t) \leq S^{spin}{}_{system}(t)$$

$$S_1(t) \geq 33$$

$$S_2(t) \geq 22$$

$$S_3(t) \geq 22$$

$$S_4(t) \geq 22$$

$$S_5(t) \geq 22$$

d) Generator Operating Reserves Requirement Constraints

$$S_1(t) + S^{oper}{}_1(t) \leq 70$$

$$S_2(t) + S^{oper}{}_2(t) \leq 100$$

$$S_3(t) + S^{oper}{}_3(t) \leq 100$$

$$S_4(t) + S^{oper}{}_4(t) \leq 60$$

$$S_5(t) + S^{oper}{}_5(t) \leq 100$$

$$S^{oper}{}_1(t) + S^{oper}{}_2(t) + S^{oper}{}_3(t) + S^{oper}{}_4(t) + S^{oper}{}_5(t) \leq S^{oper}{}_{system}(t)$$

Test Case 2 – 10-Generator, 6-Load, 10-Bus Power System

a) Branch Capacity Limit Constraints

$$|BFlow_{12,t}| \le 40$$
$$|BFlow_{23,t}| \le 30$$
$$|BFlow_{34,t}| \le 40$$
$$|BFlow_{37,t}| \le 30$$
$$|BFlow_{45,t}| \le 40$$
$$|BFlow_{56,t}| \le 40$$
$$|BFlow_{57,t}| \le 35$$
$$|BFlow_{67,t}| \le 45$$
$$|BFlow_{78,t}| \le 60$$
$$|BFlow_{89,t}| \le 30$$
$$|BFlow_{910,t}| \le 40$$
$$|BFlow_{23,t}| \le 30$$
$$|BFlow_{110,t}| \le 40$$
$$|BFlow_{24,t}| \le 30$$
$$|BFlow_{79,t}| \le 30$$

b) Generator Ramp Rate Constraints

$$S_1(t\text{-}1) - S_1(t) \le 40$$

$$S_2(t\text{-}1) - S_2(t) \le 50$$

$$S_3(t\text{-}1) - S_3(t) \le 60$$

$$S_4(t\text{-}1) - S_4(t) \le 40$$

$$S_6(t\text{-}1) - S_6(t) \le 60$$

$$S_7(t\text{-}1) - S_7(t) \leq 40$$

$$S_8(t\text{-}1) - S_8(t) \leq 50$$

$$S_9(t\text{-}1) - S_9(t) \leq 60$$

$$S_{10}(t\text{-}1) - S_{10}(t) \leq 40$$

c) Economic Maximum and Minimum Operating Constraints

$$S_1(t) + S^{spin}{}_1(t) \leq 63$$

$$S_2(t) + S^{spin}{}_2(t) \leq 90$$

$$S_3(t) + S^{spin}{}_3(t) \leq 90$$

$$S_4(t) + S^{spin}{}_4(t) \leq 54$$

$$S_5(t) + S^{spin}{}_5(t) \leq 90$$

$$S_6(t) + S^{spin}{}_6(t) \leq 63$$

$$S_7(t) + S^{spin}{}_7(t) \leq 90$$

$$S_8(t) + S^{spin}{}_8(t) \leq 80$$

$$S_9(t) + S^{spin}{}_9(t) \leq 64$$

$$S_{10}(t) + S^{spin}{}_{10}(t) \leq 34$$

$$S^{spin}{}_1(t) + S^{spin}{}_2(t) + \ldots + S^{spin}{}_9(t) + S^{spin}{}_{10}(t) \leq S^{spin}{}_{system}(t)$$

$$S_1(t) \geq 33$$

$$S_2(t) \geq 22$$

$$S_3(t) \geq 22$$

$$S_4(t) \geq 22$$

$$S_5(t) \geq 22$$

$$S_6(t) \geq 33$$

$$S_7(t) \geq 22$$

$$S_8(t) \geq 32$$

$$S_9(t) \geq 22$$

$$S_{10}(t) \geq 32$$

d) Generator Operating Reserves Requirement Constraints

$$S_1(t) + S^{oper}{}_1(t) \leq 70$$

$$S_2(t) + S^{oper}{}_2(t) \leq 100$$

$$S_3(t) + S^{oper}{}_3(t) \leq 100$$

$$S_4(t) + S^{oper}{}_4(t) \leq 60$$

$$S_5(t) + S^{oper}{}_5(t) \leq 100$$

$$S_6(t) + S^{oper}{}_6(t) \leq 60$$

$$S_7(t) + S^{oper}{}_7(t) \leq 100$$

$$S_8(t) + S^{oper}{}_8(t) \leq 100$$

$$S_9(t) + S^{oper}{}_9(t) \leq 70$$

$$S_{10}(t) + S^{oper}{}_{10}(t) \leq 100$$

$$S^{oper}{}_1(t) + S^{oper}{}_2(t) + S^{oper}{}_3(t) + \ldots + S^{oper}{}_9(t) + S^{oper}{}_{10}(t) \leq S^{oper}{}_{system}(t)$$

Test Case 3 – 50-Generator, 20-Load, 27-Bus Power System

a) Branch Capacity Limit Constraints

$$|BFlow_{12,t}| \le 40$$
$$|BFlow_{23,t}| \le 30$$
$$|BFlow_{321,t}| \le 50$$
$$|BFlow_{121,t}| \le 30$$
$$|BFlow_{45,t}| \le 40$$
$$|BFlow_{56,t}| \le 40$$
$$|BFlow_{57,t}| \le 35$$
$$|BFlow_{67,t}| \le 45$$
$$|BFlow_{78,t}| \le 60$$
$$|BFlow_{89,t}| \le 30$$
$$|BFlow_{910,t}| \le 40$$
$$|BFlow_{911,t}| \le 30$$
$$|BFlow_{110,t}| \le 40$$
$$|BFlow_{24,t}| \le 30$$
$$|BFlow_{79,t}| \le 30$$

b) Generator Ramp Rate Constraints

$$S_1(t\text{-}1) - S_1(t) \le 40$$

$$S_2(t\text{-}1) - S_2(t) \le 50$$

$$S_3(t\text{-}1) - S_3(t) \le 60$$

$$S_4(t\text{-}1) - S_4(t) \le 40$$

$$S_6(t\text{-}1) - S_6(t) \le 60$$

$$S_7(t\text{-}1) - S_7(t) \le 40$$

$$S_8(t\text{-}1) - S_8(t) \leq 50$$

$$S_9(t\text{-}1) - S_9(t) \leq 60$$

$$S_{10}(t\text{-}1) - S_{10}(t) \leq 40$$

$$S_{11}(t\text{-}1) - S_{11}(t) \leq 40$$

$$S_{12}(t\text{-}1) - S_{12}(t) \leq 50$$

$$S_{13}(t\text{-}1) - S_{13}(t) \leq 60$$

$$S_{14}(t\text{-}1) - S_{14}(t) \leq 40$$

$$S_{15}(t\text{-}1) - S_{15}(t) \leq 60$$

$$S_{16}(t\text{-}1) - S_{18}(t) \leq 40$$

$$S_{17}(t\text{-}1) - S_{17}(t) \leq 50$$

$$S_{18}(t\text{-}1) - S_{18}(t) \leq 50$$

$$S_{19}(t\text{-}1) - S_{19}(t) \leq 60$$

$$S_{20}(t\text{-}1) - S_{20}(t) \leq 40$$

c) Economic Maximum and Minimum Operating Constraints

$$S_1(t) + S^{spin}{}_1(t) \leq 63$$

$$S_2(t) + S^{spin}{}_2(t) \leq 90$$

$$S_3(t) + S^{spin}{}_3(t) \leq 90$$

$$S_4(t) + S^{spin}{}_4(t) \leq 54$$

$$S_5(t) + S^{spin}{}_5(t) \leq 90$$

$$S_6(t) + S^{spin}{}_6(t) \leq 63$$

$$S_7(t) + S^{spin}{}_7(t) \leq 90$$

$$S_8(t) + S^{spin}{}_8(t) \leq 80$$

$$S_9(t) + S^{spin}{}_9(t) \leq 64$$

$$S_{10}(t) + S^{spin}{}_{10}(t) \leq 34$$

$$S_{11}(t) + S^{spin}{}_{11}(t) \leq 63$$

$$S_{12}(t) + S^{spin}{}_{12}(t) \leq 90$$

$$S_{13}(t) + S^{spin}{}_{13}(t) \leq 90$$

$$S_{14}(t) + S^{spin}{}_{14}(t) \leq 54$$

$$S_{15}(t) + S^{spin}{}_{15}(t) \leq 90$$

$$S_{16}(t) + S^{spin}{}_{16}(t) \leq 63$$

$$S_{17}(t) + S^{spin}{}_{17}(t) \leq 90$$

$$S_{18}(t) + S^{spin}{}_{18}(t) \leq 80$$

$$S_{19}(t) + S^{spin}{}_{19}(t) \leq 64$$

$$S_{20}(t) + S^{spin}{}_{20}(t) \leq 34$$

$$S^{spin}{}_1(t) + S^{spin}{}_2(t) + \ldots + S^{spin}{}_{49}(t) + S^{spin}{}_{50}(t) \leq S^{spin}{}_{system}(t)$$

$$S_1(t) \geq 33$$

$$S_2(t) \geq 22$$

$$S_3(t) \geq 22$$

$$S_4(t) \geq 22$$

$$S_5(t) \geq 22$$

$$S_6(t) \geq 33$$

$$S_7(t) \geq 22$$

$$S_8(t) \geq 32$$

$$S_9(t) \geq 22$$

$$S_{10}(t) \geq 32$$

d) Generator Operating Reserves Requirement Constraints

$$S_1(t) + S^{oper}{}_1(t) \leq 70$$

$$S_2(t) + S^{oper}{}_2(t) \leq 100$$

$$S_3(t) + S^{oper}{}_3(t) \leq 100$$

$$S_4(t) + S^{oper}{}_4(t) \leq 60$$

$$S_5(t) + S^{oper}{}_5(t) \leq 100$$

$$S_6(t) + S^{oper}{}_6(t) \leq 60$$

$$S_7(t) + S^{oper}{}_7(t) \leq 100$$

$$S_8(t) + S^{oper}{}_8(t) \leq 100$$

$$S_9(t) + S^{oper}{}_9(t) \leq 70$$

$$S_{10}(t) + S^{oper}{}_{10}(t) \leq 100$$

$$S^{oper}{}_1(t) + S^{oper}{}_2(t) + S^{oper}{}_3(t) + \ldots + S^{oper}{}_{49}(t) + S^{oper}{}_{50}(t) \leq S^{oper}{}_{system}(t)$$