Deterministic and Stochastic Models for the Unit Commitment Problem[*]

Kashi Abhyankar[†] Tiernan Fogarty[‡] Jennifer Kimber[§] Anhua Lin[¶]
Seung Seo[||] Samer Takriti[**]

July 30, 1998

1 Introduction

Recent advances in computer technology make possible new methods for the distribution and sale of electricity. Much as the telecommunications network was deregulated, new information and monitoring systems allow for the deregulation and increased private production of electric power. Until recently, the generation and transportation of electricity were primarily controlled by a local utility company. As these utilities owned both the generators for electricity production and the distribution networks, electric power consumers were forced to buy in a sellers' market. While each state was responsible for regulating the price of electricity, each local utility essentially maintained a monopoly on electricity as a commodity.

In 1978 Congress passed the Public Utilities Regulatory Policies Act (PURPA). PURPA allowed non-utility (unregulated) companies to generate and sell electricity to regulated utility companies at the regulated price. This act was designed to increase research and development of renewable energy sources. Specifically, non-regulated companies could profit by the production of electricity by alternative means such as windmills, solar power, small dams, and co-generation plants.

More recently the prospect of complete deregulation has come about through the Energy Policy Act of 1992 and the Federal Energy Regulatory Commission's (FERC) Notice of Proposed Rule-making. While FERC has been the driving force behind electric power deregulation, individual states are responsible for erecting specific deregulation policies and electricity prices. We refer the reader to [2] and [13] for details. Figure 1 provides the current status of deregulation in each state. Comprehensive information can be found at the Department of Energy's website `http://www.eia.doe.gov/fuelelectric.html` and at the website of FERC `http://www.ferc.fed.us/intro/intro2.htm`.

[*]This work was supported in part by the Institute for Mathematics and its Applications, University of Minnesota.

[†]Department of Mathematics, University of California - Berkeley. email:`kashi@math.berkeley.edu`
[‡]Applied Math Department, University of Washington. email:`rucksack@amath.washington.edu`
[§]Department of Math and Computer Science, Kent State University. email:`jkimber@mcs.kent.edu`
[¶]Department of Mathematical Sciences, The Johns Hopkins University. email:`alin@brutus.mts.jhu.edu`
[||]Department of Mathematics, University of Minnesota. email:`sseo@math.umn.edu`
[**]Mathematical Sciences Department, IBM Watson Research Center. email:`takriti@watson.ibm.com`

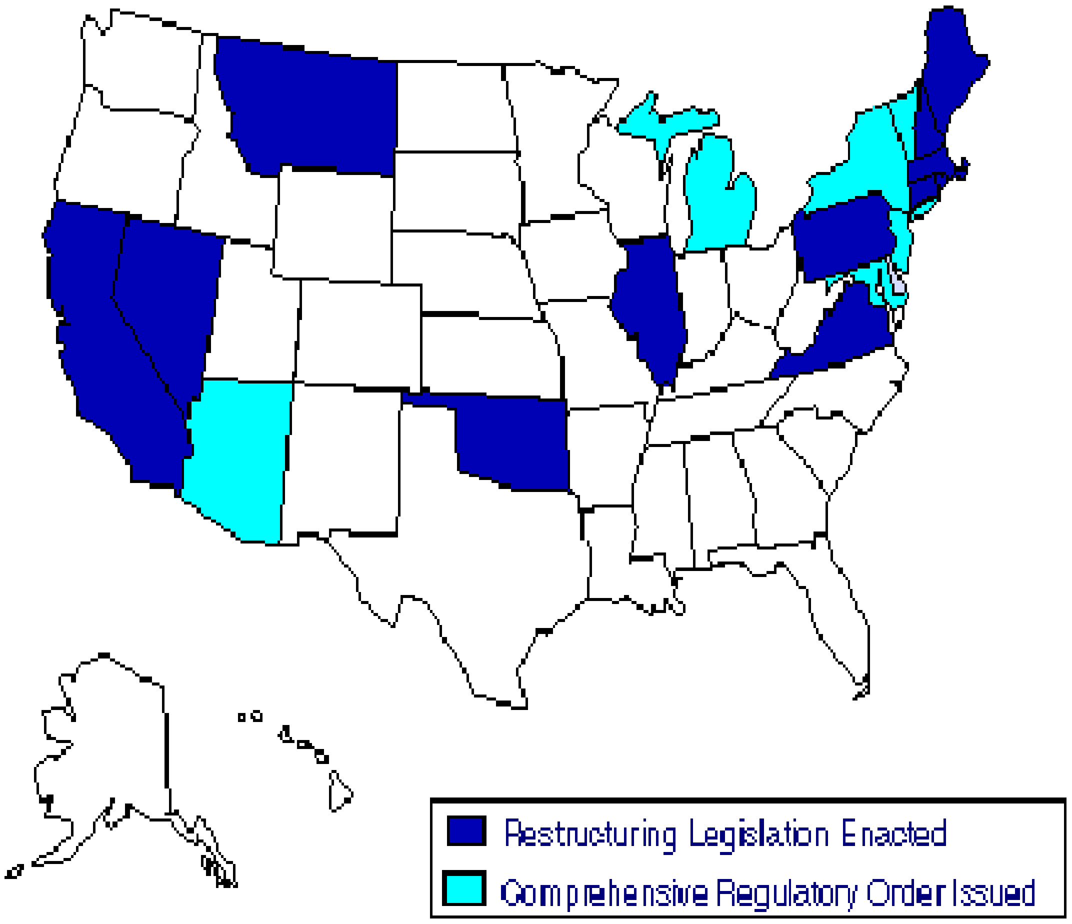
Restructuring Legislation Enacted
Comprehensive Regulatory Order Issued

Figure 1: States which have issued comprehensive deregulation orders and/or enacted restructuring legislation as of June 1, 1998.

Note that the deregulation of electricity only pertains to the generation of power and not transmission or distribution. Local utility companies still maintain a relative monopoly on the transmission and distribution of electricity since these aspects use high voltage wires and local distribution networks that utilities have already installed and own. While the generation of electricity is now a commodity on the open market, buyers are still required to purchase the transportation and distribution of power through local utility companies. Here distribution is defined as the conversion of high voltage (transmission) to lower voltage via the smaller wires linked to the buyer.

Deregulation has forced utilities to split into two types: electricity production and electricity distribution. Distribution utilities are responsible for delivering power to the consumers at the lowest possible price while guaranteeing that the demand is always met. Distribution utilities usually buy power 24 hours in advance from the lowest bidders among the production utilities. To ensure that demand will never exceed supply, distribution utilities have separate emergency contracts with generation utilities capable of producing short-term electricity at a higher cost.

Prior to deregulation, local utilities serviced a relatively constant number of customers.

With the fixed demand of a captive clientele, utilities were able to accurately forecast daily power generation needs. Profits were high as utilities were able to guarantee meeting the demand by over-producing and forcing the extra cost onto the consumer. Now that electric power generation is undergoing deregulation; consumers have the means to switch utilities at any time. For any single production utility, the change in the stability of the demand for electricity has forced expected energy demand models to change from static and deterministic to dynamic and stochastic.

In the next section, we will give details of the resources of a typical production utility and outline the constraints governing electricity production while attempting to maximize a production utility's profit. A deterministic model will be outlined for use in the case where a production utility has advance knowledge of the demand profile. We will then discuss the necessary changes in the model that must be made when future demand is stochastic. While the first model applies to a utility in a pre-deregulated region, the second (stochastic) model applies to a deregulated production utility.

2 Formulation

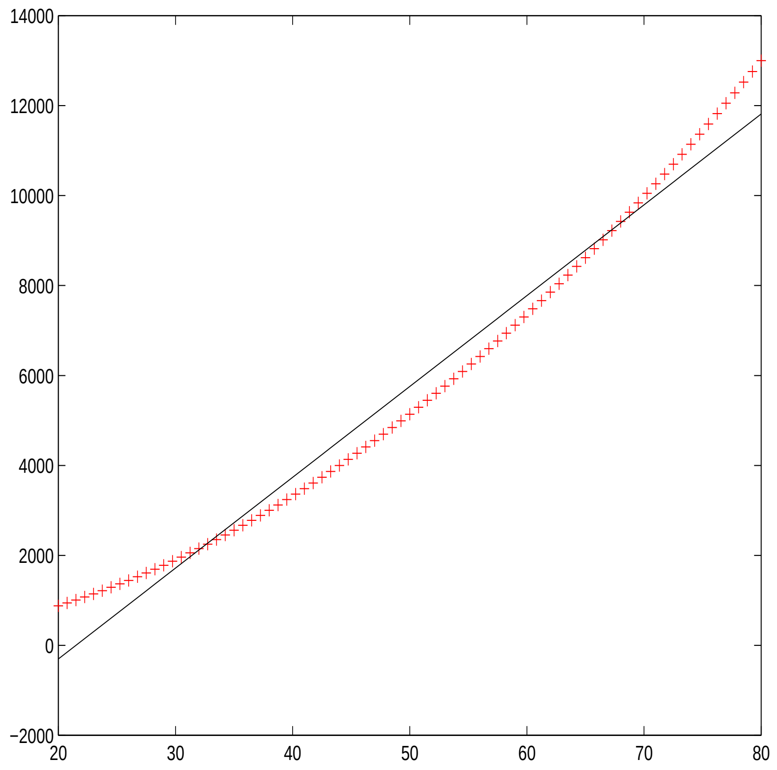

Figure 2: Quadratic function with linear least-squares fit.

In order to understand our model, a description of a typical system (generation company) must be given along with a definition of terms used to describe the model. Each utility has a certain number of generators, which can be of different types. For example, a system may have 11 coal generators, 3 nuclear generators and 8 natural gas generators. Each of these generators impose various constraints that the system must satisfy. Once turned on, a generator must be on a minimum amount of time (up time) and similarly, once turned off, a generator must remain off for a minimum amount of time (down time). In addition to the cost of running a generator (determined by the cost function), each time a generator is turned on a start up cost is incurred. While the cost function for running a generator is usually quadratic,

$$f(z) = a + bz + cz^2,$$

the coefficient c of the z^2 term is typically so small that we may approximate the quadratic cost function with a linear least-squares fit. See Figure 2 for an example of using a linear least-squares fit to approximate a quadratic with a line. Note that Figure 2 is an exaggerated example; see Figure 3 for linear fits of actual quadratic cost functions for typical generators in our study.

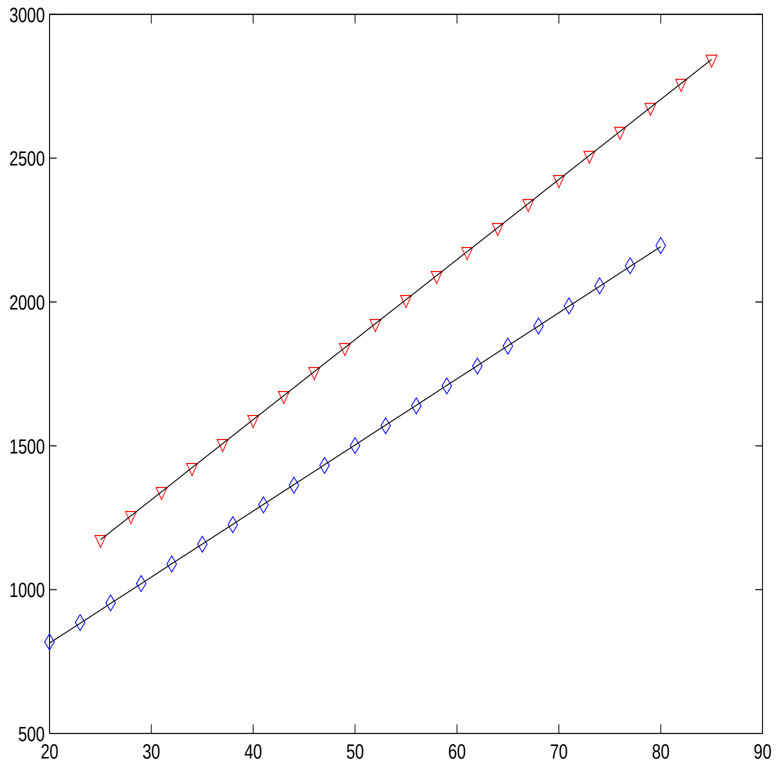

Figure 3: Quadratic cost functions of two generators ($\bigtriangledown$ and $\Diamond$) with linear least-squares fit.

2.1 Regulated Deterministic Model

In order to maximize profit, we must minimize the total cost of running the generation system. The total cost can be written as

$$\sum_{i=1}^{I} \sum_{t=0}^{T} [f_i(z_t^i) - f_i(0) + f_i(0)U_t^i + H_i G_t^i]. \tag{1}$$

Here $f_i(z_t^i)$ is the cost as a function of generated power z_t^i, where i denotes the ith generator at

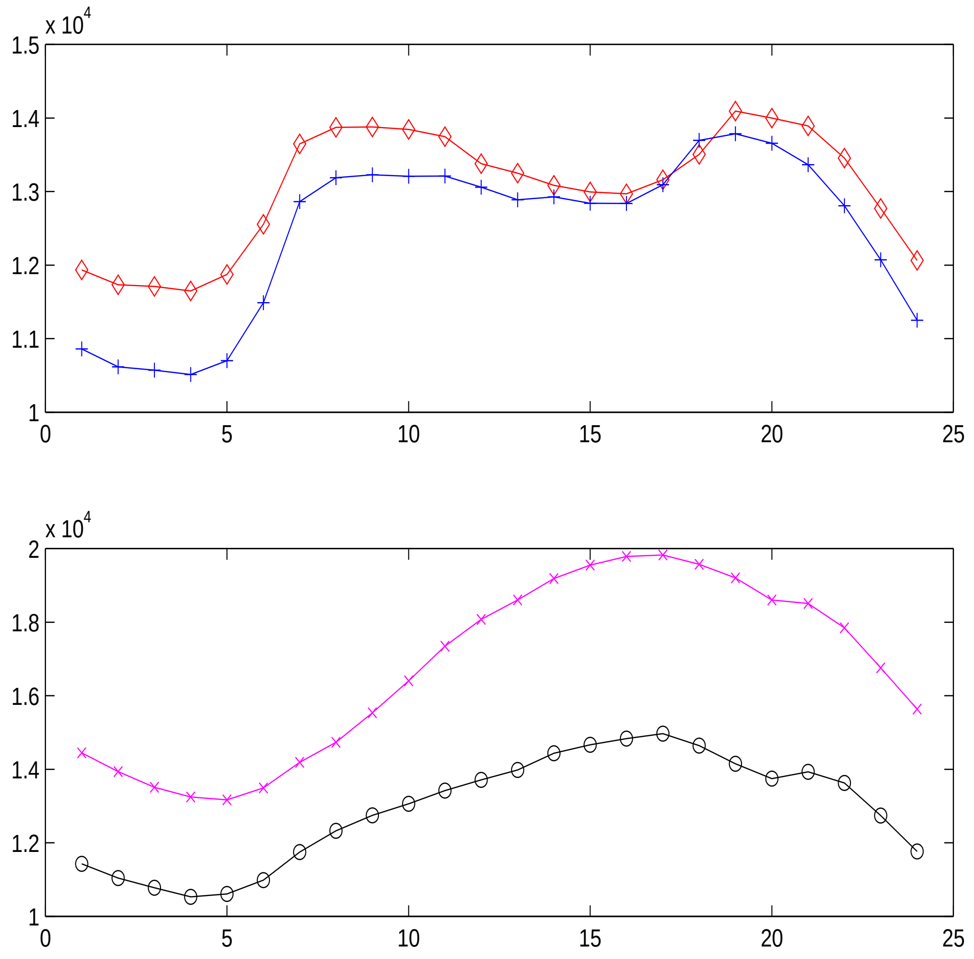

Figure 4: Sample 24 hour demand load (MWH) for February($\diamond$) and November($+$) (top figure) with May($\circ$) and August($\times$) (lower figure) of 1997 for the Southwest Power Pool.

time t. There are two binary variables, U_t^i and G_t^i, associated with each generator. If generator i is on at time t then $U_t^i = 1$, while if generator i is off at time t than $U_t^i = 0$. In addition to the cost of operation $f_i(z_t^i)$, each generator has a start-up cost H_i. Since the start-up cost H_i must be accounted for each time a generator is turned on, we set $G_t^i = 1$ if $U_{t-1}^i = 0$ and $U_t^i = 1$. The value of G_t^i is set to zero at all other times. Mathematically, this is represented as $G_t^i \geq U_t^i - U_{t-1}^i$.

The minimization is subject to the following constraints;

- The entire system of generators must produce a combined electric power supply ($\sum_{i=1}^{I} z_t^i$) which meets or exceeds the expected demand (d_t) at each hour.

$$\sum_{i=1}^{I} z_t^i \geq d_t,\ \ t = 1, \cdots, T. \tag{2}$$

Figure 4 provides examples of 24 hour demand profiles for the Southwest Power Pool (Figure 5) covering Arkansas, Kansas, Louisiana, and Oklahoma.

- Each generator has a minimum up time and a minimum down time. If a generator is turned on, then it must stay on for a minimum up time. When a generator is turned off, it must stay off for a certain down time. The minimum up and down times are determined by the type of generator. Recall that U_t^i is a binary variable so that we can specify the minimum up time requirement as

$$U_t^i - U_{t-1}^i \leq U_\tau^i, \quad \tau = t, \cdots, \min\{t + L_i - 1, T\}, t = 2, \cdots, T, \tag{3}$$

while the minimum down time can be written as

$$U_{t-1}^i - U_t^i \leq 1 - U_\tau^i, \quad \tau = t, \cdots, \min\{t + l_i - 1, T\}, t = 2, \cdots, T. \tag{4}$$

Here, L_i is the minimum up time of generator i while l_i is its minimum down time.

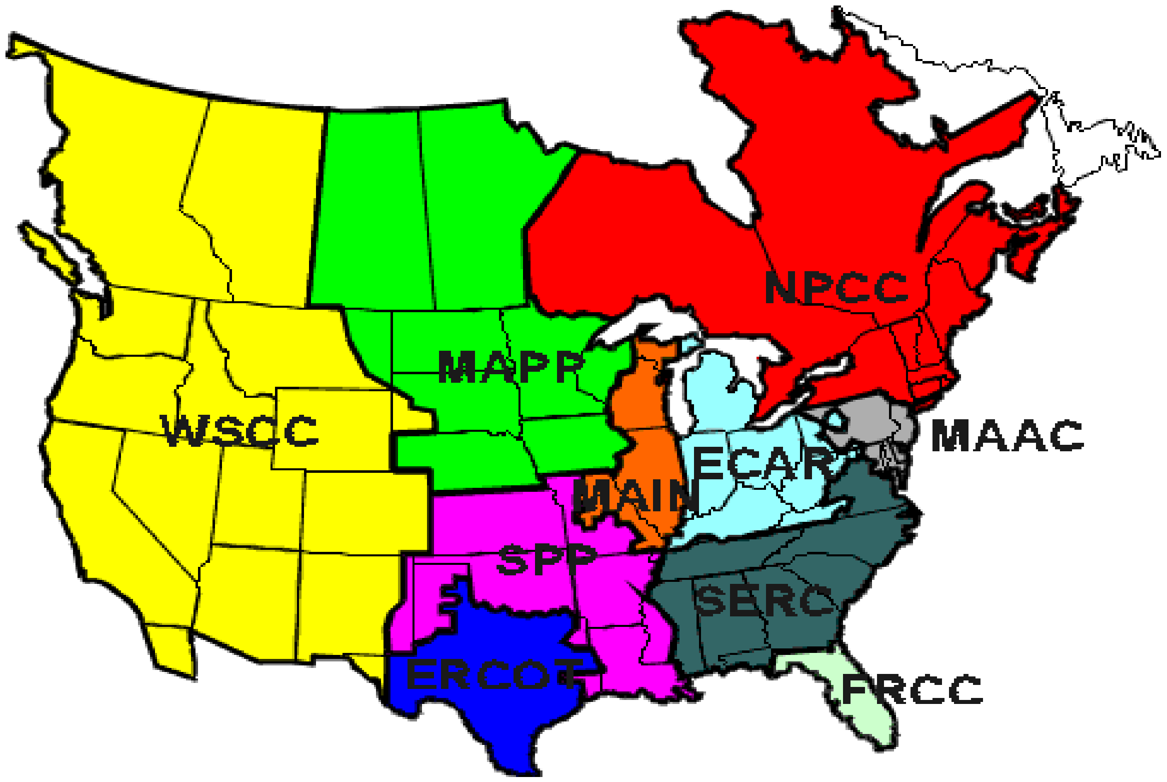

Figure 5: Electricity power pools.

- Each generator has a fixed power output range. Once a generator is turned on, it must generate a minimum amount of power q_i and can only generate a certain maximum output of power Q_i. Inside the power range, the cost is determined by the cost function f_i described above. The constraint for the power range is given as

$$q_i U_t^i \leq z_t^i \leq Q_i U_t^i, \quad t = 1, \cdots, T, \quad i = 1, \cdots, I. \tag{5}$$

- The production utility must insure that the demand is met even if a generator fails, thus we must insure that

$$\sum_{i=1}^{I} Q_i U_t^i \geq d_t + r_t. \tag{6}$$

Here r_t is known as the spinning reserves. The value assigned to spinning reserves depends on the mix of generators and demand constraints in the system.

2.2 Deregulated Stochastic Model

In a deregulated system, the distribution utility owns the transmission lines and has the responsibility of buying power from different production utilities at the lowest price. Since bidding takes place every 24 hours in advance of the expected demand, the amount of power that a given production utility must generate has become more uncertain. The stochastic nature of production forces us to alter our model to include different probabilistic future scenarios. To model uncertainty, we use a set of scenarios—sample paths—to approximate the continuous distribution of future demand. Each scenario is assigned a probability, p_s, that reflects the likelihood of its occurrence in the future. One may think of the scenarios as a discretization of the continuous distribution of electric load. The goal of the optimization becomes that of minimizing the expected cost of generating electricity while meeting the constraints imposed by the generating units and by future demand scenarios. We assume that there are S possible scenarios and denote the load by $d_{(t,s)}$, $t = 1, \ldots, T$, $s = 1, \ldots, S$.

Note that as the number of scenarios increases, the solution to our discrete model approaches that of the continuous one [1]. However, a large number of scenarios results in a large-scale mathematical model that could be numerically intractable. A good model should use a moderate number of scenarios that approximate future uncertainty well while maintaining the size of the problem under control. This is the subject of intensive research in the area of stochastic programming. In this report, we do not discuss the problem of generating scenarios. Our effort here is focused on solving the stochastic model efficiently.

Our model is a multi-stage stochastic program in which the decision maker may revise his/her decision at the end of each period of the planning horizon. Revisions are made in order to react to the load scenario being observed. For example, decisions made at the beginning of the planning horizon assume that the our knowledge of the future is summarized by the scenarios $d_{(t,s)}$ and their probabilities p_s. As time goes by, we accumulate more information regarding the system by eliminating some of the scenarios. The model should account for this ability to make decisions—recourse—as more information becomes available. To do so, we assign a set of variables, $U^i_{(t,s)}$ and $z^i_{(t,s)}$ to each scenario s.

The resulting mathematical formulation is:

$$\begin{array}{ll}
\min & \sum_{s=1}^{S} p_s \sum_{i=1}^{I} \sum_{t=1}^{T} [f_i(z^i_{(t,s)}) - f_i(0) + f_i(0)U^i_{(t,s)} + H_i G^i_{(t,s)}] \\
\text{st} & \sum_{i=1}^{I} z^i_{(t,s)} \geq d_{(t,s)}, \\
& U^i_{(t,s)} - U^i_{t-1,s} \leq U^i_{\tau}, \quad \tau = t, \cdots, \min\{t + L_i - 1, T\}, \\
& U^i_{t-1,s} - U^i_{(t,s)} \leq 1 - U^i_{\tau,s}, \quad \tau = t, \cdots, \min\{t + l_i - 1, T\}, \\
& q_i U^i_{(t,s)} \leq z^i_{(t,s)} \leq Q_i U^i_{(t,s)}, \\
& G^i_{(t,s)} \geq U^i_{(t,s)} - U^i_{t-1,s}.
\end{array} \tag{7}$$

The variables $U^i_{(t,s)}$ and $z^i_{(t,s)}$ are now functions of generator, time and possible scenario. The constraint (6), which insures against generator failure is no longer in the model as it is now the responsibility of the buyer (distribution utility) to ensure that the total demand is met. Recall that in the above models some variables are continuous while others may only take integer values. Since we have this mix of variable types (continuous and integer) our models are known as mixed-integer problems. In the next section we give a brief outline of the methods used to solve both the deterministic and stochastic mixed-integer models.

3 Solution Strategies

For the past twenty years, the unit commitment problem has been the subject of intensive research. Proposed solution methods include priority lists, dynamic programming, and Lagrangian relaxation. Priority listing is a heuristic approach in which the generating units are ranked in the order of their average costs. When demand exceeds supply, the least expensive unit is committed. Dynamic programming is an exhaustive search approach in which all possible states of the generating units are studied. As expected, dynamic programming suffers from the exponential increase in the search space as the size of the problem grows. As a result, dynamic programming is considered impractical for use in solving these types of problems.

The most successful technique for solving the unit commitment problem seems to be Lagrangian relaxation. The demand constraints are relaxed which decomposes the problem into smaller optimization problems. Each smaller problem minimizes the cost of operating a single generator. These single-generator problems can be solved easily using dynamic programming. The size of the resulting state space is relatively small which permits solving these subproblems quickly. To find an optimal solution, the Lagrangian is maximized. Given that the model is a mixed-integer program, the solution of the Lagrangian dual may not have a corresponding primal feasible solution. Furthermore, the dual function is not differentiable which complicates the problem of finding a search direction and an appropriate step size. The interested reader may refer to [7], [4], [14], and [12] for more details.

Although the unit commitment problem is a mixed-integer program, there has not been any attempt to solve it using branch-and-bound. The reason is that a large space is needed to store the search tree. Given the increase in computer speed and storage capacity over the past few years, and advancements in the field of integer programming, problems that were once considered unsolvable have been successfully solved. For example, problems arising in the airline industry, such as fleet assignment and crew scheduling, can now be handled with a reasonable degree of satisfaction. The goal of our study was to investigate the use of branch-and-bound for solving the unit commitment problem.

To solve the resulting models, we use the mixed-integer programming routine provided by CPLEX [5]. The solver is based on branch-and-bound [10]. Briefly, the integer requirement on $U^i_{(t,s)}$, $t = 1, \ldots, T$, $s = 1, \ldots, S$, $i = 1, \ldots, I$, is relaxed. The resulting linear program is solved establishing a lower bound on the optimal value of (7). If the resulting solution satisfies the integer requirement, then we stop. Otherwise, two new problems—nodes—are created. The first is constructed by adding the constraint $U^i_{(t,s)} = 0$ and the second is constructed by enforcing the condition $U^i_{(t,s)} = 1$. The process of creating new nodes is called branching. There are several methods for selecting a branching variable $U^i_{(t,s)}$. CPLEX seems to choose a variable, $U^i_{(t,s)}$, so that the term $U^i_{(t,s)}(1 - U^i_{(t,s)})$ is maximized.

After creating two new linear programs, each one is solved using the dual simplex method [9]. The node with the lowest objective value is chosen and, if needed, two new ancestors are created. The process is repeated until all nodes in the search space are exhausted. Note that CPLEX performs many operations to reduce the size of the search space. For example, if an integer solution is found, all nodes with a lower bound greater than the objective value corresponding to this feasible solution are eliminated. This process is called pruning. Another example is the case in which a near-integer solution is found. CPLEX uses heuristic techniques to construct a feasible solution. This feasible solution could be used to eliminate some of the nodes in the search tree.

Note that the branch-and-bound process maintains an upper and a lower bound on the value of the optimal objective function. These bounds get tighter as we investigate more nodes in the search tree. One may force the search process to continue until both bounds are equal—optimality. However, finding an exact optimal solution may be time consuming. This is why the search is usually stopped when the relative gap between the upper and lower bounds is below a given threshold. We use a threshold of 0.5% for most of our calculations presented in Section 4.

4 Numerical Experience

To study the effectiveness of branch-and-bound in solving the mathematical program in (7), we constructed two sets of examples. The first is based on the generation system presented in [6]. The model has 10 generators and a time horizon of 24 hours. To pass the model to CPLEX, we used AMPL [11]. The model and data files are listed in the appendix. However, we only had access to the student version of AMPL which cannot handle large models. To pass large models, we wrote a C code which could be used to create the necessary CPLEX files. The code is also listed in the appendix.

The cost functions of the generating units were assumed to be linear. The input parameters are given in Table 1. The column labeled "fix cost" provides the value of the intercept of the function f_i, while "rate" denotes the slope of f_i. Table 2 provides the electric load $d_{(t,s)}$ in MWH. The mixed-integer programming solver of CPLEX failed to solve the problem and we had to stop the process after 12 hours. We also tried a 48-hour example and CPLEX failed to solve it. As a matter of fact, CPLEX struggled with all examples if it was not given help.

To cut down the execution time and the number of searched nodes, we had to carefully study the data and provide the optimizer with the structure of the optimal solution. We list some of the approximating assumptions that we used in our calculations.

1. It was clear that the first generator had a lower cost than any other generating unit. Furthermore, its capacity was lower than the minimum load over the 24-hour planning horizon. As a result, one can set $U^1_{(t,s)} = 1$ for all $t = 1, \ldots, T$ and $s = 1, \ldots, S$.

2. Another trick was finding units that are similar and forcing the optimizer to commit them one at a time. For example, units 3 and 4 have similar properties. The optimizer may study a large number of nodes trying to decide which unit to choose over the other. By choosing to dominate generator 3 with generator 4, the execution time was cut significantly.

3. One can determine if a unit dominates another. This is the case if two units i and j have similar minimum up and down times, and if one of them, i, has a smaller start-up and running costs, then we can impose the constraints $U^i_{(t,s)} \geq U^j_{(t,s)}$, $t = 1, \ldots, T$, $s = 1, \ldots, S$.

4. Finally, to reduce the effect of T, we solved the problem as a sequence of optimization problems. For example, when $T = 24$, we solved the problem as if the horizon was

generator	min load	max load	min on	min off	fix cost	rate	start cost
1	150	455	8	8	1000	16.19	4500
2	150	455	8	8	970	17.26	5000
3	20	130	5	5	700	16.6	550
4	20	130	5	5	680	16.5	560
5	25	162	6	6	450	19.7	900
6	20	80	3	3	370	22.26	170
7	25	85	3	3	480	27.74	260
8	10	55	1	1	660	25.92	30
9	10	55	1	1	665	27.27	30
10	10	55	1	1	670	27.79	30

Table 1: Generator properties of example 1.

Hours	1	2	3	4	5	6	7	8	9	10	11	12
Demand	700	750	850	950	1000	1100	1150	1200	1300	1400	1450	1500
Hours	13	14	15	16	17	18	19	20	21	22	23	24
Demand	1400	1300	1200	1050	1000	1100	1200	1400	1300	1100	900	800

Table 2: Electric demand of example 1 in MWH

16 hours only. Then, the resulting solution was used to fix the value of $U^i_{(t,s)}$ for the first 8 hours, $t = 1, \ldots, 8$, and for all units and scenarios. Then, the next 16 hours, $t = 9 \ldots, 24$, were solved, and so on. This approach seems to work quite well with all of our test problems.

We attempted to solve each test problem twice. In the first attempt, the model was passed on to CPLEX with some restrictions on the order in which different units could be committed. In the second run, we shortened the planning horizon and used a rolling-window approach to solve the problem. The results of our numerical tests are summarized in Table 3. The first row provides the results for the case in which the full horizon was considered, while the second row provides the results when the planning horizon was broken into sequences of 16-hour windows. For the case of 10 generators, we used 1, 3, and 9 scenarios. The scenarios branched at hours 8 and 16.

The last two rows of Table 3 contain the results for a generation system with 32 generating units. This system was too large for CPLEX and we had to fix the status of many of the generating units. Fixing the status of some units is acceptable from a practical point of view. The generating units are usually split into must-run, cyclers, and peakers. The variables $U^i_{(t,s)}$ corresponding to the must-run units could be set to one. For peakers, these variables could be set to zero. In practice, a peaker can be started in 5–10 minutes and there is no need to schedule its use hours in advance. As a result, the model contained 18 generating units—cyclers.

In all cases, we were able to achieve an error of less than 0.5% when using the above-mentioned approximations. When the full problem—without breaking the horizon—was solved, the error was approximately three times higher.

generator	scenario	upper bound	lower bound	% error	nodes	time
10	1	544392	543656	0.135	2011	103.55
		543923	543746	0.032	870	24.9
10	3	551456	542144	1.688	2170	515.24
		545671	542933	0.501	914	227.69
10	9	551682	541776	1.795	719	1080.95
		545511	542656	0.523	2090	1881.58
32	9	1120510	1120501	0.001	38	2470.23
		1127096	1124295	0.248	217	577.4

Table 3: Branch-and-bound applied to four test problems.

5 Conclusions

As the electric power industry deregulates electricity generation, power producers are faced with higher uncertainty in their electric load and prices. Stochastic modeling of these systems is becoming increasingly important. While the stochastic models can provide well-hedged solutions, they cause serious difficulties from an optimization point of view. These models are often large mixed-integer programs and solving them requires effective approximating techniques. We attempted to solve these models using CPLEX and managed to attain accurate results. However, the maximum number of scenarios in our test runs was nine; barely sufficient to approximate future uncertainties in a highly volatile market. More work is needed to develop better methods for solving such models efficiently.

References

[1] J. R. Birge and F. Louveaux. *Introduction to Stochastic Programming.* Springer Verlag, New York, 1997.

[2] Mississippi Public Service Commissio. Revised prosed transition plan for retail competition in the electric industry, June 1998. Post Office Box 1174, Jackson, MS. 39215-1174.

[3] Darinka Dentcheva and W. Rőmisch. Optimal power generation under uncertainty via stochastic programming. Technical report, Hűmboldt-Universitat Berlin, Institut fűr Mathematik, 1998.

[4] D. P. Bertsekas G. S. Lauer, N. R. Sandell Jr. and T. A. Posbergh. Solution of large-scale optimal unit commitment problems. *IEEE Transactions on Power Apparatus and Systems*, 101(1):79–86, 1982.

[5] CPLEX Optimization Inc. *Using the CPLEX Callable Library*, 1995.

[6] S. A. Kazarlis, A. G. Bakirtzis, and V. Petridis. A genetic algorithm solution to the unit commitment problem. *IEEE Transactions on Power Systems*, 11(1):83–92, February 1996.

[7] A. Merlin and P. Sandrin. Anew method for the unit commitment at electricite de france. *IEEE Transactions on Power Apparatus and Systems*, 102(2):1218–1225, 1983.

[8] A. Mőller and W. Rőmisch. A dual method for the unit commitment problem. Technical report, Hűmboldt-Universitat Berlin, Institut fűr Mathematik, 1995.

[9] K. G. Murty. *Linear Programming.* John Wiley & Sons, 1983.

[10] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization.* John Wiley & Sons, Inc, 1988.

[11] D. M. Gay R. Fourer and B. W. Kernighan. *AMPL: A Modeling Language For Mathematical Programming.* boyd & fraser publishing company, 1993.

[12] G. B. Sheblé and G. N. Fahd. Unit commitment literature synopsis. *IEEE Transactions on Power Systems*, 1994.

[13] Penny Tvrik. Iowa utilities board investigates electric wheeling. *CIRAS News*, 30(2), winter 1996.

[14] A. J. Wood and B. F. Wollenberg. *Power Generation, Operation, and Control.* John Wiley, 1996.

Appendices

AMPL Model

To implement our model, we used AMPL [11]. The following is the AMPL model file for the deterministic unit commitment problem.

```
param time;
param gen;

param demand {t in 1..time};
param c_low {i in 1..gen};
param c_upp {i in 1..gen};
param fix {i in 1..gen};
param l_on {i in 1..gen};
param l_off {i in 1..gen};
param startc {i in 1..gen};
param rate {i in 1..gen};

var Z {i in 1..gen, t in 1..time} >= 0;
var U {i in 1..gen, t in 1..time} binary;
var G {i in 1..gen, t in 1..time} >= 0;

minimize total_cost:
   sum{i in 1..gen, t in 1..time}
    (rate[i]* Z[i,t]+ fix[i] * U[i,t] + startc[i]*G[i,t]);
```

```
subject to lowCap {i in 1..gen, t in 1..time}:
      c_low[i] * U[i,t] <= Z[i,t] ;

subject to upCap {i in 1..gen, t in 1..time}:
      Z[i,t] <= c_upp[i]* U[i,t];

subject to meetDemand {t in 1..time}:
      sum{i in 1..gen} Z[i,t] >= demand[t];

subject to start {i in 1..gen, t in 2..time}:
      G[i, t] >= U[i,t] - U[i,t-1];

subject to keepon {i in 1..gen, t in 2..time, r in t..min(t+l_on[i]-1, time)}:
      U[i,t] - U[i,t-1] <= U[i,r];

subject to keepoff {i in 1..gen, t in 2..time,r in t..min(t+l_off[i]-1, time)}:
      U[i,t-1] - U[i,t] <= 1 - U[i,r];

# Constraints that could help the solver in achieving better performance

subject to fixUnit {t in 1..time} :
   U[1,t] = 1;
subject to sixOverSeven {t in 1..time}:
   U[6,t] >= U[7,t];
subject to fourOverThree {t in 1..time}:
   U[4,t] >= U[3,t];
```

The following is the data file for a problem with 10 generators, 24 hour horizon, and one scenario

```
param gen := 10;
param time := 24;

param:
   l_on   l_off   c_low   c_upp   startc   fix   rate :=
 1    8       8     150     455     4500  1000  16.19
 2    8       8     150     455     5000   970  17.26
 3    5       5      20     130      550   700  16.60
 4    5       5      20     130      560   680  16.50
 5    6       6      25     162      900   450  19.70
 6    3       3      20      80      170   370  22.26
 7    3       3      25      85      260   480  27.74
 8    1       1      10      55       30   660  25.92
 9    1       1      10      55       30   665  27.27
10    1       1      10      55       30   670  27.79;
```

```
param: demand :=
 1    700
 2    750
 3    850
 4    950
 5   1000
 6   1100
 7   1150
 8   1200
 9   1300
10   1400
11   1450
12   1500
13   1400
14   1300
15   1200
16   1050
17   1000
18   1100
19   1200
20   1400
21   1300
22   1100
23   900
24   800;
```

C Code

Unfortunately, we only had access to the student edition of AMPL. To pass the large models to CPLEX, we wrote a C code that helped in creating the input files needed to run CPLEX. Here is our code.

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

#define GENERATORS 20
#define PERIODS 72
#define BRANCHES 81
#define NAME 1024
#define FIXPERIODS 12

/* Format a string, change the spaces in a string into 0 */

void format (char string[])
{
  char *ptr;
```

```
   for (ptr=string; *ptr!=0; ptr++)
     if (*ptr == ' ')
       *ptr = '0';
}


/* Read the information of the generators  */

long readGenerator (char name[],
    double minCapacity[],
    double maxCapacity[],
    long   minUpTime[],
    long   minDoTime[],
    double intercept[],
    double slope[],
    double startUp[])
{
  long gen;
  FILE *fp;

  fp = fopen (name, "r");
  if (fp == NULL)
    {
      printf ("-> Could not open %s.\n", name);
      exit (EXIT_SUCCESS);
    }

  for (gen=0; gen<GENERATORS; gen++)
    if (fscanf (fp, "%lf%lf%ld%ld%lf%lf%lf",
&minCapacity[gen], &maxCapacity[gen],
&minUpTime[gen], &minDoTime[gen],
&intercept[gen], &slope[gen],
&startUp[gen]) == EOF)
      break;

  fclose (fp);

  printf ("-> The model has %ld generators.\n", gen);
  return (gen);
}

/* Read the information of loads   */

double**  readLoad (char name[],
    long* periods,
    long* branches,
    double probability[]
    )

{
  long period,num_of_periods,num_of_branches;
  long branch;
  int i;
  double ** load;
  FILE *fp;

  fp = fopen (name, "r");
  if (fp == NULL)
    {
      printf ("-> Could not open %s.\n", name);
      exit (EXIT_SUCCESS);
    }


  fscanf(fp,"%ld%ld",&num_of_branches,&num_of_periods);

  /* allocate memory for storing the load data   */
  load=(double **)malloc(sizeof(double *)*num_of_branches);
  for(i=0;i<num_of_branches;i++)
    load[i]=(double *)malloc(sizeof(double)*num_of_periods);

  /* read the load data   */
  for(branch=0;branch<num_of_branches;branch++)
    {

      fscanf(fp,"%lf",&probability[branch]);


      for(period=0;period<num_of_periods;period++)
fscanf(fp,"%lf",&load[branch][period]);

    }

  fclose(fp);

  *branches=num_of_branches;
  *periods=num_of_periods;

  printf ("-> The model has %ld periods\n", period);

  return load;
}
```

```
/* Write the cost about binary variables */

void writeBinaryObjective (const long   periods,
                           const long   generators,
                           const long   num_of_branches,
   const double intercept[],
                           const double startUp[],
                           const double probability[],
   FILE   *fp)
{
  char string[NAME];
  long generator, period,branch;

  fprintf (fp, "Minimize \n");

  for (branch=0;branch<num_of_branches;branch++)
    {

      for (generator=0; generator<generators; generator++)
{
  for (period=0; period<periods; period++)
    {
      sprintf (string, "u%2ld%3ld%2ld", generator, period,branch);
      format (string);
       fprintf (fp, " + %lf %s\n",
probability[branch]*intercept[generator], string);
    }
  fprintf (fp, "\n");
}
    }

   for(branch=0;branch<num_of_branches;branch++)
     {

       for (generator=0; generator<generators; generator++)
 {
   for (period=0; period<periods; period++)
     {
       sprintf (string, "v%2ld%3ld%2ld", generator, period,branch);
       format (string);
       fprintf (fp, " + %lf %s\n",
probability[branch]*startUp[generator], string);
     }
   fprintf (fp, "\n");
 }
```

```
     }

}

/* Write the linear cost into output file  */

void writeLinearObjective (const long   periods,
                           const long   generators,
                           const double slope[],
   const double probability[],
   const long num_of_branches,
   FILE   *fp)
{
  char string[NAME];
  long generator, period,branch;

  for(branch=0;branch<num_of_branches;branch++)
    {
      for (generator=0; generator<generators; generator++)
{
  for (period=0; period<periods; period++)
    {
      sprintf (string, "x%2ld%3ld%2ld", generator, period,branch);
      format (string);
      fprintf (fp, " + %lf %s\n",
       probability[branch]*slope[generator], string);
    }
  fprintf (fp, "\n");
}
    }

  fprintf (fp, "st\n");
}

/* Write the constraints of minimum up time into output file  */
void writeMinUpTime(const long periods,
    const long generators,
    const long minUpTime[],
    const long num_of_branches,
    FILE *fp)
{
  char periodString[NAME],previousString[NAME],tauString[NAME];
  long generator,period,branch;
  long tau;

  for(branch=0;branch<num_of_branches;branch++)
```

```
    {

      for  (generator=0; generator<generators; generator++)
for (period=1; period<periods; period++)
  {
    sprintf (periodString, "u%2ld%3ld%2ld", generator, period,branch);
    format (periodString);
    sprintf (previousString, "u%2ld%3ld%2ld", generator,
     period-1,branch);
    format (previousString);

    for (tau=period; tau<period+minUpTime[generator] && tau<periods;
 tau++)
      {
sprintf (tauString, "u%2ld%3ld%2ld", generator, tau,branch);
format (tauString);
fprintf (fp, "%s - %s - %s <= 0\n", periodString,
 previousString, tauString);
      }
  }
    }

}

/* Write the constraints of minimum down time into output file */

void writeMinDoTime (const long periods,
                     const long generators,
                     const long minDoTime[],
                     const long num_of_branches,
     FILE *fp)
{
  char periodString[NAME], previousString[NAME], tauString[NAME];
  long generator, period,branch;
  long tau;

  for(branch=0;branch<num_of_branches;branch++)
    {
      for  (generator=0; generator<generators; generator++)
for (period=1; period<periods; period++)
  {
    sprintf (periodString, "u%2ld%3ld%2ld",
     generator, period,branch);
    format (periodString);

    sprintf (previousString, "u%2ld%3ld%2ld",
     generator, period-1,branch);
    format (previousString);

    for(tau=period; tau<period+minDoTime[generator] && tau<periods;
tau++)
      {
sprintf (tauString, "u%2ld%3ld%2ld", generator, tau,branch);
format (tauString);
fprintf (fp, "%s - %s + %s <=  1\n",
 previousString, periodString, tauString);
      }
  }
    }
}

/* Write the constraints of start_up variables  */

void writeStartUp (const long   periods,
                   const long   generators,
                   const double startUp[],
                   const long   num_of_branches,
   FILE   *fp)
{
  char string[NAME], periodString[NAME], previousString[NAME];
  long generator, period,branch;

  for (branch=0;branch<num_of_branches;branch++)


    for (generator=0; generator<generators; generator++)
      {
for (period=1; period<periods; period++)
  {
    sprintf (periodString, "u%2ld%3ld%2ld",
     generator, period,branch);
    format (periodString);

    sprintf (previousString, "u%2ld%3ld%2ld",
     generator, period-1,branch);
    format (previousString);

    sprintf (string, "v%2ld%3ld%2ld",
     generator, period, branch);
    format (string);

    fprintf (fp, "%s - %s - %s <= 0",
     periodString, previousString, string);

    fprintf (fp, "\n");
  }
      }

}

/* Write the constraints of min,max capacities into output file  */

void writeBounds (const long   periods,
                  const long   generators,
                  const double minCapacity[],
                  const double maxCapacity[],
                  const long   num_of_branches,
  FILE *fp)
{
  char uString[NAME], xString[NAME];
  long generator, period,branch;

  for (branch=0;branch<num_of_branches;branch++)
    {

      for  (generator=0; generator<generators; generator++)
for (period=0; period<periods; period++)
  {
    sprintf (uString, "u%2ld%3ld%2ld", generator, period,branch);
    format (uString);
    sprintf (xString, "x%2ld%3ld%2ld", generator, period,branch);
    format (xString);

    fprintf (fp, "%lf %s - %s <= 0.0\n",
     minCapacity[generator], uString, xString);
    fprintf (fp, "%lf %s - %s >= 0.0\n",
     maxCapacity[generator], uString, xString);
  }
    }
}

/* Write the constraints of demand into output file  */

void writeDemand (const long   periods,
                  const long   generators,
  double** load,
                  const long   num_of_branches,
  FILE   *fp)
```

```
{
  char string[NAME];
  long generator, period,branch;

  for(branch=0;branch<num_of_branches;branch++)
    for (period=0; period<periods; period++)
      {

for (generator=0; generator<generators; generator++)
  {

    sprintf (string, "x%2ld%3ld%2ld",
     generator, period,branch);
    format (string);
    fprintf (fp, " + %s", string);
    if (generator%6==0 && generator>1) fprintf(fp,"\n");
  }

fprintf (fp, ">= %lf\n", load[branch][period]);

      }

}

/* The common parts of different scenarios should have the same value */

void writeEqual(const long periods,
const long generators,
double ** load,
const long num_of_branches,
FILE * fp)
{
  char String1[NAME],String2[NAME],String3[NAME],String4[NAME];
  long generator,period,branch1,branch2,num_of_equal;
  double temp;
  long i;

  for(period=0;period<periods;period++)
    {
      branch1=0;branch2=1;num_of_equal=0;

      while(branch1<num_of_branches)
{

  /* fix branch1,move branch2 until it's corresponding load
     value is different from the one for branch1  */
  while(branch2<num_of_branches &&
load[branch2][period]==load[branch1][period])
    {
      num_of_equal++;
      branch2++;
    }

  /* at this period,the load value of branch1 is unique  */
  if (num_of_equal==0)
    {
      branch1++;
      branch2=branch1+1;
    }

  /* at this period,branch1 shares the load value with
     num_of_equal other branches   */
  else
    {
      for(generator=0;generator<generators;generator++)
{
  sprintf(String1,"u%2ld%3ld%2ld",generator,
  period,branch1);
  format (String1);

  sprintf(String3,"x%2ld%3ld%2ld",generator,
  period,branch1);
  format(String3);

  for (i=branch1+1;i<branch2;i++)
    {
      sprintf(String2,"u%2ld%3ld%2ld",generator,
      period,i);
      format(String2);

      sprintf(String4,"x%2ld%3ld%2ld",generator,
      period,i);
      format(String4);

      fprintf(fp,"%s - %s =0\n",String1,String2);

      fprintf(fp,"%s - %s =0\n",String3,String4);

    }
}
      branch1=branch2;
      branch2++;
      num_of_equal=0;
    }
}
    }
}

/* Write additional constraints according to the analysis of the
   generator's information into the output file  */

/* this function should be changed according to different data of
   generators                    */

void writeAddConstraints (const long periods,
  const long generators,
  const long scenarios,
  FILE *fp)
{
  long period, generator, scenario,generatorA,generatorB;
  char string[NAME],stringA[NAME],stringB[NAME];

  /* The first generator is always on */

  generator = 1;
  for (period=0; period<periods; period++)
    for (scenario=0; scenario<scenarios; scenario++)
      {
sprintf (string, "u%2ld%3ld%2ld", generator-1, period, scenario);
format (string);

fprintf (fp, "%s = 1\n", string);
      }


  /* Six is better than seven */
  generatorA=6;generatorB=7;

  for (period=0;period<periods;period++)
    for (scenario=0;scenario<scenarios;scenario++)
      {
sprintf (stringA,"u%2ld%3ld%2ld",generatorA-1,period,scenario);
format(stringA);
sprintf (stringB,"u%2ld%3ld%2ld",generatorB-1,period,scenario);
format(stringB);

fprintf(fp, "%s - %s >= 0\n",stringA,stringB);
```

```
      }


  /* Three is better than four */

  generatorA=3;generatorB=4;

  for (period=0;period<periods;period++)
    for (scenario=0;scenario<scenarios;scenario++)
      {
sprintf (stringA,"u%2ld%3ld%2ld",generatorA-1,period,scenario);
format(stringA);
sprintf (stringB,"u%2ld%3ld%2ld",generatorB-1,period,scenario);
format(stringB);

fprintf(fp, "%s - %s >= 0\n",stringA,stringB);
      }
}

/* Fix the u variables of the first several periods   */

void fix_period(FILE *fp,char name[],long generators,long num_of_branches,
 long num_of_fix_periods)
{
  char c[8] ,a ,string[NAME],u0;
  double v;
  long i,j,k,u[GENERATORS][FIXPERIODS][BRANCHES];
  FILE * fpr;
  int generator,period,scenario;

  fpr=fopen(name,"r");
  if (fpr==NULL)
    {
      printf("->Could not open %s .\n",name);
      exit(EXIT_SUCCESS);
    }

  /* set all the u variables of the first several periods to 0  */
  for(i=0;i<generators;i++)
    for(j=0;j<num_of_fix_periods;j++)
      for(k=0;k<num_of_branches;k++)
u[i][j][k]=0;

  /* set part of the u variables to 1  */
  while(1)
    {
      fscanf(fpr,"%c%2ld%3ld%2ld%lf%c",&u0,&generator,&period,&scenario,
     &v,&a);
      if (u0!='u') break;
      else
if (period<num_of_fix_periods)
  u[generator][period][scenario]=1;
    }

  /* write the constraints into output file */
  for (i=0;i<generators;i++)
    for(j=0;j<num_of_fix_periods;j++)
      for(k=0;k<num_of_branches;k++)
{
  sprintf(string,"u%2ld%3ld%2ld",i,j,k);
  format(string);
  if (u[i][j][k]==0)
    fprintf(fp,"%s = 0.000000\n",string);
  else fprintf(fp,"%s = 1.000000 \n",string);
}

  fclose(fpr);
}

/* All the u variables are binary numbers   */

void writeInteger (const long   periods,
                   const long   generators,
                   const long   num_of_branches,
   FILE *fp)
{
  char uString[NAME];
  long generator, period,branch;

  fprintf (fp, "integers\n");

  for (branch=0;branch<num_of_branches;branch++)
    for  (generator=0; generator<generators; generator++)
      for (period=0; period<periods; period++)
{
  sprintf (uString, "u%2ld%3ld%2ld", generator, period,branch);
  format (uString);
  fprintf (fp, "%s\n", uString);
}

  fprintf (fp, "end\n");
```

```
}

/* main function  */

void main (void)
{
  char name[NAME];
  long generators, periods,num_of_branches,fix_periods;
  long minUpTime[GENERATORS], minDoTime[GENERATORS];
  double minCapacity[GENERATORS], maxCapacity[GENERATORS],
  intercept[GENERATORS], slope[GENERATORS],
  startUp[GENERATORS],
  probability[BRANCHES];
  double** demand;
  int answer;

  FILE *fp;

  printf ("-> Enter the name of the generators file: ");
  scanf ("%s", name);

  /* read information of generators    */
  generators = readGenerator (name, minCapacity, maxCapacity,
minUpTime, minDoTime, intercept,
slope, startUp);

   printf ("-> Enter the name of the demand file: ");
   scanf ("%s", name);

   /* read information of load      */
   demand = readLoad (name,&periods,&num_of_branches,probability);

   printf ("-> Enter the name of the output file: ");
   scanf ("%s", name);
  /* fp points to the output file   */
  fp = fopen (name, "w");


  /* write the cost about binary variables into output file  */
  writeBinaryObjective (periods, generators, num_of_branches,
 intercept, startUp, probability,fp);

  /* write linear cost into output file            */
  writeLinearObjective (periods, generators, slope, probability,
 num_of_branches,fp);
```

```
  /* write the constraints of minimum up time into output file  */
  writeMinUpTime (periods, generators, minUpTime,num_of_branches, fp);


  /* write the constraints of minimum down time into output file */
  writeMinDoTime (periods, generators, minDoTime,num_of_branches, fp);

  /* write the constraints about start_up variables    */
  writeStartUp (periods, generators, startUp,num_of_branches, fp);

  /* write the constraints about min,max capacities    */
  writeBounds (periods, generators, minCapacity, maxCapacity,
num_of_branches,fp);


  /* write the constraints about demand               */
  writeDemand (periods, generators, demand,num_of_branches, fp);


  /* the common parts of different scenarios have same values  */
  writeEqual (periods, generators, demand, num_of_branches, fp);

  /* write additional constraints according to analysis of
     the information of generators                     */
  writeAddConstraints (periods, generators, num_of_branches, fp);


  printf("\n do you need to fix the u values of the first several
          periods ?(1=yes  or 0=no)\n ");
  scanf ("%d",&answer);

  if (answer!=0)
    {
      printf("\n how many periods you want to fix :");
      scanf ("%ld",&fix_periods);

      printf ("\n->Enter the name of the solution file for the
                first %ld periods:",fix_periods);

      scanf ("%s", name);

      fix_period(fp,name,generators,num_of_branches,fix_periods);

    }
  /* all the u values are binary numbers         */
  writeInteger (periods, generators,num_of_branches, fp);


  fclose (fp);
}
```