STONY BROOK UNIVERSITY

CEAS Technical Report 837

Phase Balancing Algorithms:

Kai Wang, Steven Skiena and Thomas G. Robertazzi

October 11, 2012

Phase Balancing Algorithms

*Kai Wang, **Steven Skiena, *Thomas G. Robertazzi

*Electrical Engineering Department, Stony Brook University
100 Nicolls Road, Stony Brook, 11790 USA
Corresponding author: Kai Wang. Tel: 1-631-6627528
Email address: kaiwang.sunysb@gmail.com

**Computer Science Department, Stony Brook University
100 Nicolls Road, Stony Brook, 11790 USA

Abstract

Unbalanced loads on feeders increase power system investment and operating costs. Single-phase lateral loads phase swapping is one of the popular methods to balance such systems. In this paper, six algorithms for phase balancing are studied, including a Genetic Algorithm, Simulated Annealing, a Greedy Algorithm, Exhaustive Search, Backtracking algorithm and a Dynamic Programming algorithm. The novel dynamic programming algorithm in particular produces optimal solutions for this NP-complete problem efficiently.

Keywords: Phase Balancing, Dynamic Programming, Genetic Algorithm, Simulated Annealing, Greedy algorithm, Backtracking algorithm.

1. Introduction

Over the past 15 years, research has been conducted on three phase feeder balancing. Phase balancing aims to reduce the unbalance of loads on three phases which can bring severe voltage drops in the feeders. The majority of electric power systems utilize, in the electric distribution system, feeders which carry three phases of alternating current/voltage. It is desirable for electric utilities and providers of electric power distribution systems to have approximately equal loads on each phase. This is a problem as even if loads are initially balanced, with time loads increase, decrease, are added or removed from each phase, causing an unbalance of loads. Even during the same day there may be much variation of load on each phase of a feeder. There are two major phase balancing methods: there is feeder reconfiguration at the system level and there is phase swapping at the feeder level [1]. Phase swapping is not as well developed in the literature as feeder reconfiguration. This paper is about phase swapping algorithms.

Why does one wish phases to be in balance? Phase unbalance can limit the amount of power transferred on a feeder as on an unbalanced feeder one phase may reach its maximum carrying capacity measured in amperes (i.e. ampacity) while the other two phases are then underutilized and unable to

carry their full or even nearly their full amount of current. This is poor utilization of the existing power distribution network and may result in unnecessary feeder expansion and upgrades which raise utility costs. Because one phase may be near its maximum ampacity, phase unbalance can also lead to preventive breaker/relay tripping and shutdown of a feeder whose restoration also involves a cost to the electric utility.

Periodically crews rebalance feeders. This can be done during periods of maintenance or restoration. One suburban Northeast U.S. utility rebalances feeders if the percentage of unbalance exceeds 15%. Generally it takes 10 to 15 minutes to switch a load so the overall job may take an hour plus travel time to the location. Work by a crew of two employees can cost several hundred dollars. However preparatory work such as scheduling can bring the total cost to several thousand dollars for one tap change. Three factors are considered in making a decision to rebalance a feeder: the monetary cost of making the tap change(s), the expected increase in feeder balance and the temporary interruption of power to the customer. Tap change generally fall into two situations: a new customer is to be connected or the phase balance for existing feeders has become significantly unbalanced. Once a feeder is re-balanced it will initially be in balance but drift into unbalance as time

goes on.

Even in more limited electric power systems, the same problems may arise. For instance Gaffney [2] reports problems with effective phase balancing in electric power systems in the tactical battlefield environment, largely because of insufficient operator training and experience. David [3] [4] proposes automatic phase balancing but does not propose an algorithm for this purpose.

The variables in the phase balancing problem are the phases each load is connected to and the goal is to minimize the degree of unbalance on feeders. Several algorithms have been used to solve phase balancing problem. In 1999 Zhu, Bilbro and Chow introduced simulated annealing to the problem [1]. In 2000 and 2004, Chen and Cherng and Gandomkar applied a genetic algorithm to the problem [5] [6]. In 2007, Huang, Chen, Lin, et. al used an immune algorithm to solve the problem [7].

Many combinatorial optimization problems have no known efficient algorithms capable of always producing optimal solutions. For those problems that computer scientists have been shown to be NP-complete, there is convincing evidence that no correct, efficient algorithms can exist. An efficient algorithm for any one of the hundreds of known NP-complete problems would

imply efficient algorithms for all of them, implying that all are equally hard to compute.

The phase balancing problem we describe in this paper can readily be shown to be equivalent to integer partitioning, a well-known NP-complete problem. Thus an efficient algorithm for phase balancing which always produced optimal solutions would imply efficient algorithms for all problems in NP, which computer scientists considered extremely unlikely. However heuristic algorithms that produce near optimal solutions with reasonable efficiency are possible, and are often developed for this purpose. [8]

In this paper, six algorithms are applied to the phase balancing problem: Exhaustive search and a Backtracking algorithm(section 2). Greedy Algorithm (section 3), Simulated Annealing (section 4), Genetic Algorithm (section 5) and Dynamic Programming (section 6). Selected parameterized illustrations of the use of each algorithm appear in each algorithm discussion/ section. A systematic comparison of the algorithms is made in section 7. The conclusion appears in section 8.

We purposefully did not consider particle swarm optimization (PSO) and differential evolution (DE) algorithms in this paper. We note that these methods are most appropriate for complex problems with ill-defined search

spaces, as opposed to classical combinatorial optimization problems like ours, which is essentially a variant of the knapsack problem.

The key lesson of our paper is that heuristic techniques such as simulated annealing and genetic algorithms (and DE and PSO) are superseded by the dynamic programming and combinatorial search methods we employ, which give optimal results instead of heuristic ones. The primary lesson of this paper is that our new dynamic programming algorithm gives optimal results in reasonable time.

2. Objective function and stopping criteria

There are various kinds of objective functions such as cost functions in [1] and the loss function in [5]. Also, loads could be connected to two phases. Here we consider that all the loads are connected to single phase. The load range is set as integers between 1 and 100. Larger loads range can be scaled to this range. In this test, the objective function is the phasing unbalance index (PUI) which is used in many phase balancing papers [7] [9] [10]:

$$PUI = \frac{Max(|I_a - I_{avg}|, |I_b - I_{avg}|, |I_c - I_{avg}|)}{I_{avg}} * 100\% \qquad (1)$$

Here, I_a, I_b and I_c are the current on phase 1, 2 and 3. I_{avg} is the mean

value of the current on each single phase.

The stopping criteria is: when the objective value reaches 1/500 times of the initial value. Other stopping criterias are of course possible.

3. Exhaustive search and a backtracking algorithm

For n loads, each load is connected to one phase throughout the paper. Since there are n loads, and each load can be on one of three phases, there are 3^n ways to assign the loads to different phases. Under exhaustive search we calculate minimum objective functions for any potential number of tap changes. One then selects the solutions which satisfy the stopping criteria and finds the minimum number of tap changes among them.

Here we present a backtracking algorithm which can obtain an optimal solution as exhaustive search does, but has a smaller computational complexity [8]. Suppose one wants the most balanced solution using at most t tap changes. This can be done with backtracking in $O((2n)^t)$ since each tap change has two phase choices, which is better than the exhaustive search for small t. In this solution, the state space will be a vector of length t: The candidates for the ith position will be the possible tap changes greater than the last one in terms of load indexes.

The pseudo code appears below. Here S_k is the set of candidate nodes in the decision tree for k tap changes:

Backtrack - DFS(A,k)

if $A = (a_1, a_2, \cdots, a_k)$ is a solution, report it.

else

$k = k + 1$

compute S_k

while $S_k \neq \oslash$ do

a_k = an element in S_k

$S_k = S_k - a_k$

Backtrack - DFS(A,k)

4. Greedy Algorithm

4.1. Greedy Algorithm

A greedy algorithm is any algorithm that finds a local optimal solution at every step. It gives a global optimal solution to many problems, but not all problems. It does not give a globally optimal solution to the phase balancing problem.

4.2. Greedy Algorithm for Phase Balancing

The steps are:

1. Input the loads and initial phases.

2. Calculate total loads on each phase.

3. Select one load from the phase with largest total load and move it to the phase with smallest total load. The load is selected so it can minimize the difference of the total loads on those two phases.

4. Calculate the objective value and see if it satisfies the stopping criteria. If yes, finish. If not, return to step 3.

4.3. Results

In figure 2, 100 randomly generated loads and phases are used for testing. The figures are the average of 300 runs. The horizontal axis is the number of tap changes the program needs and the vertical axis is the probability they appeared in 300 runs.

5. Simulated Annealing

5.1. Simulated Annealing Algorithm

The intuition behind the simulated annealing algorithm comes from the process of molten metals. The system is slowly cooled in order to achieve its

lowest energy state. The basic idea of the method is that, in order to avoid being trapped in local minima, the algorithm usually accepts a "move" to a better solution but occasionally accepts a "move" that worsens the objective function with probability of:

$$Prob_{Accept} = e^{-\triangle E/kT} \tag{2}$$

Here e is the irrational number (=2.71828...). $\triangle E$ is the change between the objective values for two different solutions, k is a constant relationship between temperature and energy, and T is "temperature".

Simulated annealing is applicable to problems where one solution can be transformed into another by a "move" and there is an objective function available for evaluating the quality of a solution.

5.2. Simulated Annealing Algorithm for Phase Balancing

The steps are:

1. Input the loads and initial phases. Calculate the objective value.

2. Randomly select one load, move its phase to a randomly selected phase. Calculate the objective value.

3. Calculate the difference between the values in previous two steps. If the difference is negative, accept the "move". If not, accept the move with

the probability in equation 2.

4. Repeat step 2 until it meets the stopping criteria.

5.3. Results

In figure 3, 100 randomly generated loads and phases are used for testing. The horizontal axis is time axis, the vertical axes are objective values and numbers of tap changes. It can be seen that beyond a certain number of tap changes there is only a minimal improvement on the objective function (law of "diminishing returns").

6. Genetic Algorithm

6.1. Genetic Algorithm

Genetic algorithms belong to a larger family of algorithms known as evolutionary algorithms. They apply concepts from the theory of biological evolution, such as natural selection, reproduction, genetic diversity and propagation, species competition/cooperation, and mutation, to search and optimization problems.

A genetic algorithm starts from the initial population (initial phases of the loads from some random solution) which are represented by a string of

binary numbers. In each generation, crossover, mutation and selection are applied to the population in order to converge to the best solution.

6.2. Coding

In GA, a population is a set of solutions (chromosomes) for the objective function. In the population, the variables are encoded by use as binary numbers. For phase balancing problem, 2 bits are considered since one needs to represent 3 phases. "00", "01" and "10" represent phase 1, 2 and 3 respectively. After mutation or crossover, "11" will be changed to "00", "01" and "10" with equal probability if it is generated.

6.3. Objective function

In the GA process, the value of the objective function mirrors the property of a solution. Better solutions have larger objective values.

6.4. Crossover

The crossover operator will proceed as follows. A crossover point is selected randomly for each of two solutions. A crossover probability is then invoked (P_c from 0.6 to 0.8) to decide whether to make a swap of bits. An example is shown as follow:

String 1: 110|**10011**

String 2: 101|**01110**

Crossover point

String 1: 101|**01110**

String 2: 101|**10011**

6.5. Mutation

The mutation operation is implemented by randomly selecting any binary bit with a prespecified probability (about 0.01) and reversing it. The purpose of mutation in *GA*s is preserving and introducing diversity. Mutation should allow the algorithm to avoid local minima by preventing the population of chromosomes from becoming too similar to each other, thus slowing or even stopping evolution.

An example is shown as follow:

Before: 11010011

Mutation point

After: 11000011

6.6. Selection

The selection operator creates new populations or generations by selecting individuals from the old population. The selection is probabilistic but biased

towards the best solutions as special deterministic rules are used. In the new generations created by the selection operator, there will be more copies of the best individuals and fewer copies of the worst. A common technique for implementing the selection operator is the roulette wheel approach.

In this process, the individuals of each generation are selected for survival into the next generation according to a probability value proportional to the ratio of individual fitness (i.e. value of objective function) over total population fitness; this means that on average the next generation will receive copies of an individual in proportion to the importance of its fitness value.

6.7. Results

As shown in figure 4 and 5. 100 random generated loads and phases are used for testing. The figures are the average of 300 runs. In first graph, the horizontal axis is the number of generations for the genetic algorithm and the vertical axis is the corresponding objective value. In second graph, the horizontal axis is the number of tap changes the program needs and the vertical axis is the probability they appeared in 300 runs.

We also simulated an immune algorithm which is similar to genetic algorithm in the use of crossover, mutation and selection. It was found to perform similar to our genetic algorithm.

7. Dynamic Programming

7.1. A Dynamic Programming Algorithm to Solve the Phase Balancing Problem

An "optimal" algorithm for phase balancing is now presented. The phase balancing problem is NP-complete even with two phases and no cost per tap change, because it is equivalent to the integer partition problem and the integer partition problem is NP-complete. The hardness of integer partition depends upon large numbers, because it is not strongly NP-complete. For the phase balancing problem, the loads range between 1 and several thousand amperes. Assume that there are n loads, where the ith load has weight w_i and is currently assigned to feeder l_i. We assume the weights of all loads are integers, and the total load $T = \sum_{i=1}^{n} w_i$. As will be seen the algorithm runs faster with smaller T. Loads can be scaled to bring this about. The solution produced by the dynamic programming algorithm are optimal but it should be noted that the scaling is a source of approximation.

We present an algorithm which runs in $O(nT^2)$ to find the minimum number of changes to reach a particular quality criteria.

Denote the total load on phase i by L_i. Because there are 3 phases, there are about T^2 sets of possible values for L_1, L_2, and L_3. This is as both L_1

and L_2 are integers between 0 and T, and $L_3 = T - L_1 - L_2$. L_3 would be specified after one has L_1 and L_2.

The algorithm will enumerate all possible partitions of T into L_1, L_2, and L_3, and in particular for each such partition P find way to move from the current state to P using the fewest number of changes. One can evaluate each of these $O(T^2)$ partitions according to the objective function, eliminate all which are not good enough, and then find the minimum cost good-enough transformation.

Define $C[x, y, i]$ to be the minimum cost (in terms of number of moves) to realize a balance of $L_1 = x, L_2 = y$ and implicitly $L_3 = T - L_1 - L_2$ after reassignments to the first i loads (from 1 to i).

We define the following recurrence relation:

$$C[x, y, i] = Min[C[x-l_i, y, i-1]+t(i, 1), C[x+l_i, y-l_i, i-1]+t(i, 2), C[x+l_i, y, i-1]+t(i, 3)] \tag{3}$$

Here $t(i, l)$ is the cost of moving the i_{th} load to phase l. $C[x, y, i]$ is the minimum number of tap changes to move from the initial loads to $[x, y, T_i - x - y]$.

$$T_i = \sum_{j=1}^{i} L_j \tag{4}$$

If i_{th} load stays on phase l

$$t(i, l) = 0 \tag{5}$$

If i_{th} load leaves phase l

$$t(i, l) = 1 \tag{6}$$

Assume the ith load is initially on line 1. Then the optimal solution either leaves load i on line 1 (incurring no cost for the move), or moves it to line 2, or moves it to line 3 (both of which incur a cost of 1 operation). We need similar recurrences for the cases where load i is on line 2 or line 3. The basis of this recurrence is that $C[L_1, L_2, 0] = 0$, $C[x_0, y_0, 0] = \infty$ for all $x_0 \neq L_1$ and $y_0 \neq L_2$ (meaning no other states are achievable with zero moves).

Lastly, one calculates objective values for all $(T_n + 1)^2$ possible $[x, y, n]$ using equation 1 and get the minimum tap changes from $C[x, y, n]$. Thus, one does not need to calculate objective values for other $C[x, y, i]$, $i \in [1, n-1]$ which significantly reduces running time. In other words, the dynamic programming algorithm naturally produces a minimal solution for all number of tap changes desired, efficiently.

7.2. Results

As in figure 6, 100 randomly generated loads and phases are used for testing. The figure is the average of 300 runs. The horizontal axis is the number of tap changes the program recommends and the vertical axis is the probability they appeared in 300 times running.

8. Comparison

Two factors affect the results: the objective value and the number of tap changes. The objective value represents the unbalance of the loads on feeders. Meanwhile, each tap change costs some amount of money. So the aim is to get the desired objective value with a number of tap changes which is as small as possible.

Table 1 compares the performance of the six approaches in terms of running time and the number of tap changes for a single run of each program written in *Matlab*. Table 2 illustrates the performance improvement of each algorithm. The table shows wins (W), losses (L) and ties (T) of the algorithm in the first column compared to the algorithm listed for each column.

From the previous two tables, one can see that DP and GA are probably the better algorithms for the Phase Balancing problem. So to further com-

pare the performance of DP and GA, we have two tables. Table 3 and table 4 are two tables made from 20 examples of DP. Every example has 10 loads and the integer load range is 1 to 10. The columns differ in the number of tap changes. The rows are the different runs. The numbers in the table are the objective values. Table 5 is made from 20 examples of the genetic algorithm, every one with the same initial loads and phases as DP has. The vertical axis is the number of runs. The first column holds the best objective values that the GA obtained and the second column includes the corresponding number of tap changes the GA needed for those objective values. From these two tables, one can see that in some cases the best objective values that the two algorithms can get are same and DP needs less tap changes and other times GA loses both in terms of the objective value and number of tap changes.

Another test as in figure 7 is a gathering of 30 examples. In each example, both the DP and the GA are give same set of loads and initial phases and then we record the result they give in terms of the number of tap changes. Every example has 10 loads and the integer load range is 1 to 10. DP lost very few times but DP gave better objective values in those examples.

Since most of the time DP and GA have the same performance in terms of the unbalance factor, but GA needs a larger number of tap changes, it was

desired to investigate the size of the loads those two algorithm transferred. that is. whether GA transfers more loads with relatively smaller size and DP transfers loads with relatively larger size. Figure 8 and figure 9 show the loads' size VS the frequency that were transferred. From figure 8 and figure 9, one can see that DP is somewhat biased to larger values of switched load size and the loads that GA transfers are distributed evenly in load size because GA transfers loads randomly but DP transfers loads with less tap changes (less total load transfer). While most of the algorithms in the paper were implemented in $Matlab$, running time of a faster C version of the dynamic programming algorithm appears in Table 6.

9. Conclusion

Dynamic programming does the best in terms of performances though it is the slowest of the non-exhaustive algorithms. This is a very promising algorithm because of the algorithm's optimality. It was found that even though the genetic algorithm and dynamic programming produced solutions that were almost identical in terms of the unbalance factor to many significant places, the genetic algorithm can require many more tap changes than dynamic programming did (often by more than a factor of two). This

suggests that the solution space contains a variety of optimal/near-optimal solutions that differ significantly in the number of tap changes. The genetic algorithm, at least as presently constituted, is not able to discern the best solution as well as the dynamic programming algorithm. The good news is that an optimal algorithm for phase balancing in dynamic programming is available with reasonable complexity ($O(nT^2)$). This is an interesting problem from both the viewpoint of algorithms and an interesting power system application.

Acknowledgment

This material is based upon work supported by the Department of Energy under Award Number DE-OE0000220.

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, pro process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process,

or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

This report was prepared as an account of work performed by The Research Foundation of SUNY as sub-recipient of any award made by an agency of the United States Government to the Long Island Power Authority. Neither the Long Island Power Authority nor any of its trustees, employees or subsidiaries, nor the State of New York, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring the Long Island Power Authority, its trustees or employees, or by the Sate of New York. The views and opinions of authors expressed herein do not necessarily state or reflect those of the Long Island Power Authority, its trustees or employees.

or of the State of New York.

References

[1] J. Zhu, G. Bilbro and M. Chow. IEEE Transactions on Power Systems. "Phase Balancing using Simulated Annealing". Vol. 14, No. 4. November 1999, pp. 1508-1513.

[2] M. N. Gaffney. http://www.ndia-mich.org/workshop/Papers, "Intelligent Power Management: Improving Power Distribution in the Field".

[3] Y. David and R. Hasharon. United States Patent. "Apparatus for and method of evenly distributing an electrical load across an N-phase power distribution network". US Patent Num. 6018203. Jan. 25. 2000.

[4] Y. David. et al. United States Patent. "Apparatus for and method of evenly distributing an electrical load across a three phase power distribution network". US Patent Num. 5604385. Feb. 18. 1997.

[5] M. Gandomkar. 39th International Universities Power Engineering Conference. "Phase Balancing Using Genetic Algorithm". Sept. 2004. pp. 377-379.

[6] T. H Chen and J. T. Cherng. IEEE Transactions on Power Systems. "Optimal Phase Arrangement of distribution Transformers Connected to a Primary Feeder for System Unbalance Improvement and Loss Reduction Using a Genetic Algorithm". Vol. 15, NO. 3, August 2000, pp 994-1000.

[7] M.-Y. Huang, C.-S. Chen, C.-H. Lin, M.-S. Kang, H.-J. Chuang and C.-W. Huang. IET Generation, Transmission and Distribution, "Three-phase balancing of distribution feeders using immune algorithm". 17th August 2007, pp. 383-392.

[8] Steven Skiena. "The Algorithm Design Manual" 2nd edition. Springer. 2008.

[9] Chia-Hung Lin, Chao-Shun Chen, Hui-Jen Chuang and Cheng-Yu Ho. IEEE Transactions on Power Systems, "Heuristic rule-based phase balancing of distribution systems by considering customer load patterns". VOL. 20, NO. 2, May 2005. pp 709-716.

[10] Nikhil Gupta, Anil Swarnkar and K. R. Niazi. Power and Energy Society General Meeting, 2011 IEEE. "A novel strategy for phase balancing in three phase four wire distribution systems". July 2011.

Table 1: Performance comparison

Algorithms	Running Time (ms)
Exhaustive Search	6642.97
Backtracking Algorithm	886.49
Greedy Algorithm	5.65
Simulated Annealing	1.45
Genetic Algorithm	72.15
Dynamic Programming	223.62

Table 2: Performance comparison 2

Algorithm	DP	GA	Greedy	SA
DP		10W/40T	42W/8T	45W/5T
GA	10L/40T		40W/6T/4L	42W/4T/4L
Greedy	8T/42L	4W/6T/40L		40W/4T/6L
SA	39L/11T	4W/4L/42T	6W/4T/40L	

Table 3: Objective function and number of tap changes for dynamic programming for ten runs

Number of tap changes:	1	2	3
1st Run	0.2750	0.0500	0.0500
2nd Run	0.0179	0.0179	0.0179
3rd Run	0.0678	0.0169	0.0169
4th Run	0.0500	0.0500	0.0500
5th Run	0.2558	0.0465	0.0465
6th Run	0.4769	0.1077	0.0154
7th Run	0.1053	0	0.0526
8th Run	0.1321	0.0755	0.0189
9th Run	0.6154	0.2000	0.0154
10th Run	0.1077	0.0154	0.0154

Table 4: Objective function and number of tap changes for dynamic programming for ten runs

Number of tap changes:	4	5	6
1st Run	0.1250	0.2750	0.8000
2nd Run	0.0714	0.1786	0.4464
3rd Run	0.0678	0.0169	0.2203
4th Run	0.0500	0.0500	0.0500
5th Run	0.0154	0.0615	0.2462
6th Run	0.0154	0.0615	0.2462
7th Run	0.0526	0.2105	0.3684
8th Run	0.0189	0.1887	0.4151
9th Run	0.0154	0.1538	0.2462
10th Run	0.0615	0.1077	0.2000

Table 5: Objective function and number of tap changes for genetic algorithm for ten runs

	Objective value	Number of tap changes
1st Run	0.0500	5
2nd Run	0.0714	5
3rd Run	0.0169	7
4th Run	0.0500	5
5th Run	0.0465	5
6th Run	0.0615	5
7th Run	0.0526	5
8th Run	0.0755	5
9th Run	0.1077	5
10th Run	0.1077	5

Table 6: Running time for different number of loads with DP method in C program

Number of loads	10	50	100
Running time (sec)	≈ 0	1	5

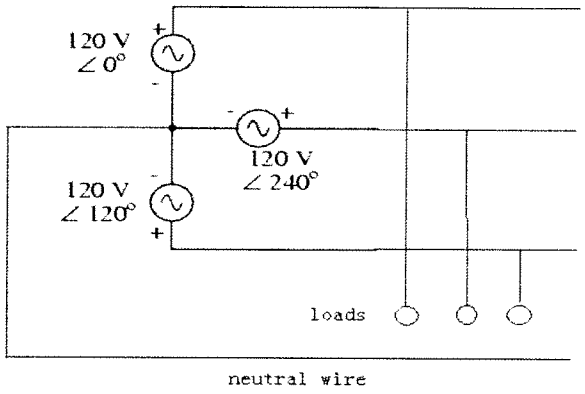

Figure 1: Three phase wiring diagram

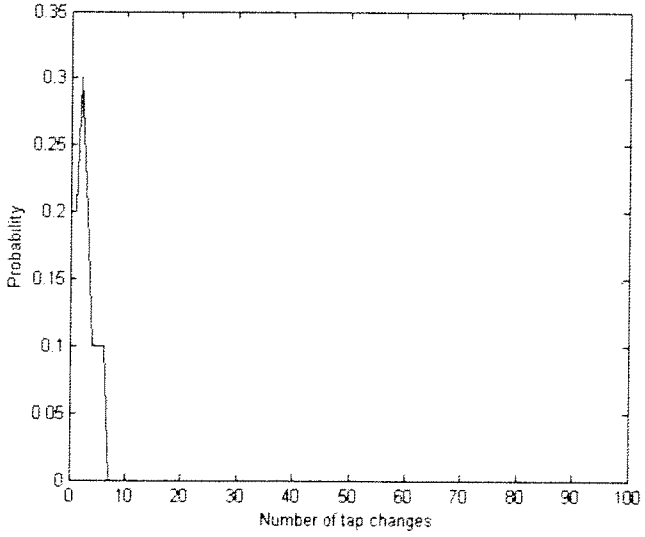

Figure 2: Probability VS Number of tap changes for the greedy algorithm

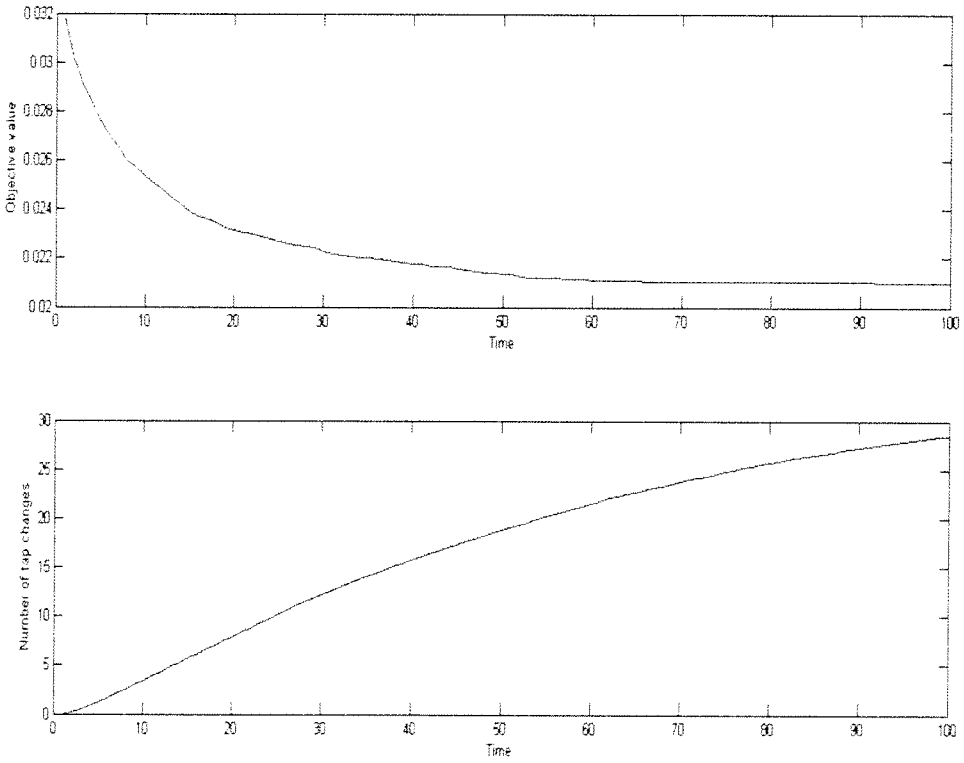

Figure 3: Probability VS Number of tap changes for the simulated annealing

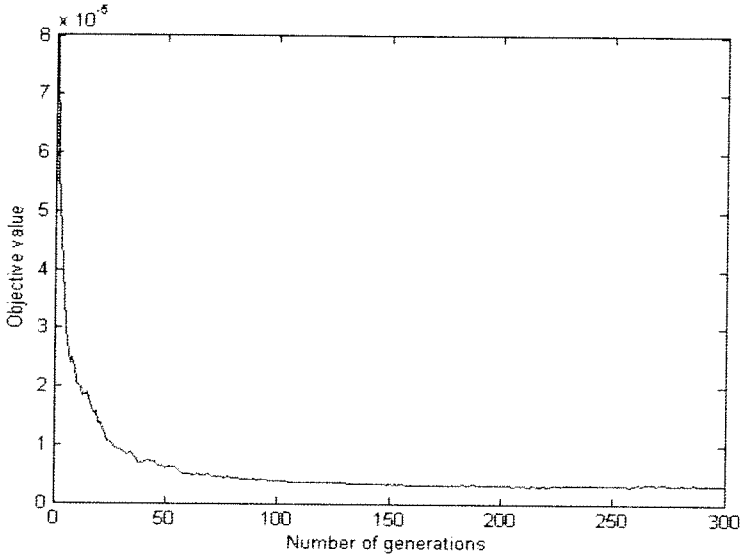

Figure 4: Fitness value VS Number of generations for a genetic algorithm

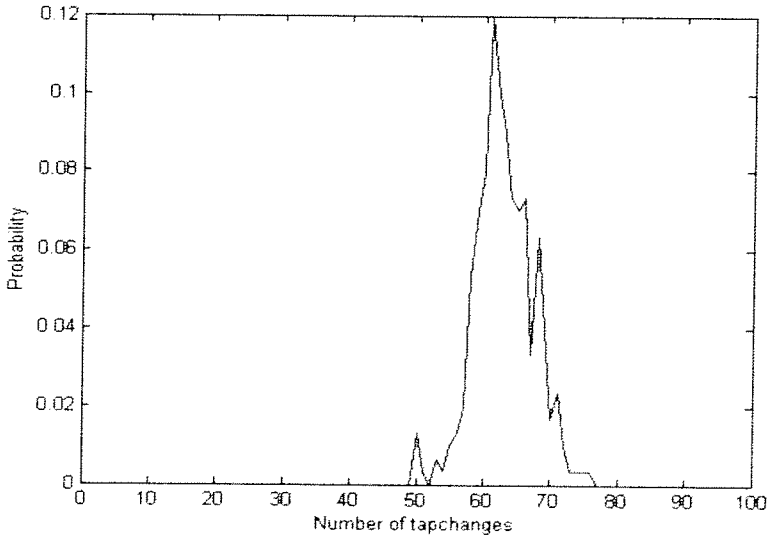

Figure 5: Probability VS Number of tap changes for a genetic algorithm

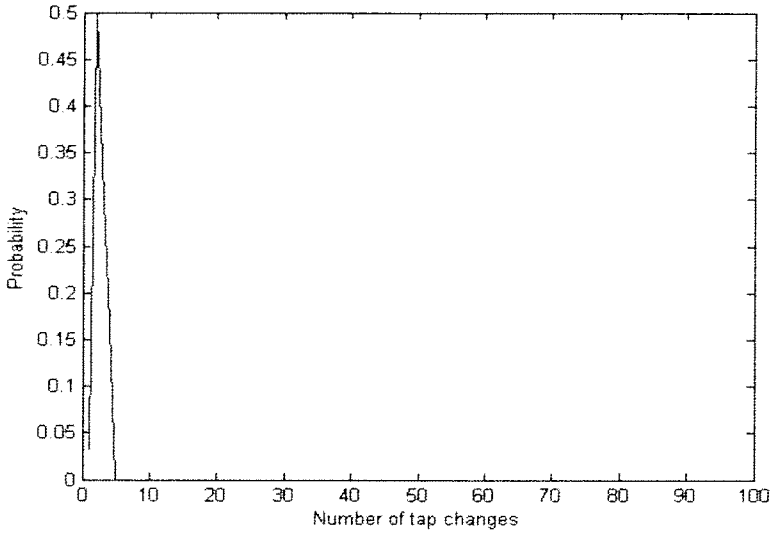

Figure 6: Probability VS Number of tap changes for the dynamic programming

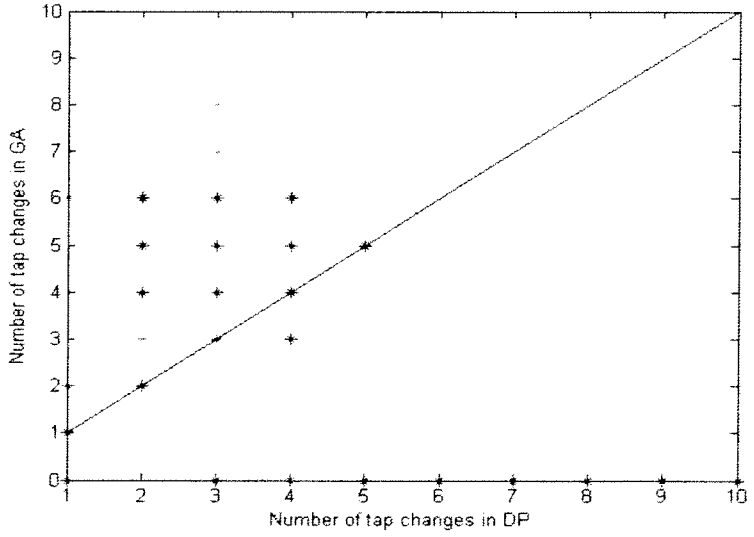

Figure 7: Comparision of number of tap changes between DP and GA

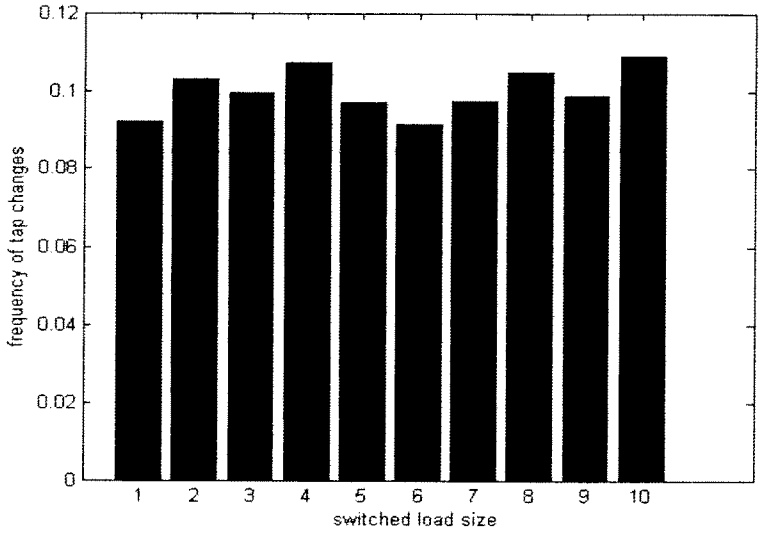

Figure 8: Tap change load size VS frequency for genetic algorithm

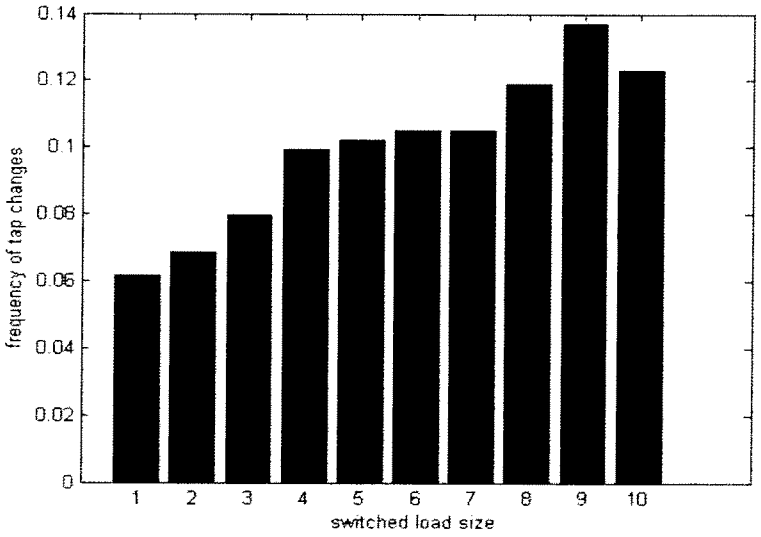

Figure 9: Switched load size VS frequency for dynamic programming