Metaheuristic Algorithms for Task Assignment in Distributed Computing Systems: A Comparative and Integrative Approach

Peng-Yeng Yin*, Benjamin B.M. Shao, Yung-Pin Cheng and Chung-Chao Yeh

Department of Information Management, National Chi Nan University, 303 University Rd., Puli, Nantou 545, Taiwan

Abstract: We consider the assignment of program tasks to processors in distributed computing systems such that system cost is minimized and resource constraints are satisfied. Several formulations for this task assignment problem (TAP) have been proposed in the literature. Most of these TAP formulations, however, are NP-complete and thus finding exact solutions is computationally intractable. Recently, some approximation methods like simulated annealing have been proposed, and simulation results exhibited the potential to solve the TAP using metaheuristics. In order to better understand the strengths and weaknesses of various metaheuristics applied to the TAP, we first propose two alternative metaheuristics—one using genetic algorithm and the other reinforcement learning algorithm—as well as their implementation details. Extensive computational evidences of the two heuristic algorithms against that of simulated annealing are presented, compared and discussed. Based on these experimental results, a hybrid strategy employing both metaheuristics is then proposed in order to solve the TAP more effectively and efficiently.

Keywords: Computer, heuristic algorithms, task assignment problem, distributed systems, genetic algorithms, reinforcement learning, simulated annealing.

1. INTRODUCTION

A distributed computing system is defined as a collection of computers interconnected by a telecommunication network that attempts to disperse the data processing function and fits the needs of modern decentralized organization structures. Besides the capability of implementing a logically integrated information system for geographically dispersed corporations, distributed computing systems provide other benefits, such as quick access to data, higher system reliability, and ease of incremental growth [1].

On the other hand, efficient utilization of resources in distributed computing systems is also important [2]. In distributed computing systems, it is characteristically difficult to assign the tasks of a program application to distributed processors such that a certain measure of system costs is minimized and system resources are effectively utilized. Several formulations of this task assignment problem (TAP) have been proposed in the literature to cope with various types of system costs and environmental constraints. In general, the TAP is NP-complete and finding exact solutions is computationally prohibitive [3].

In addressing this intractability issue, previous endeavors of TAP research can be classified into three areas. First, exact mathematical programming approaches using column generation [4] and branch-and-bound [5-6] have been proposed. Second, efficient algorithms have been developed for solving TAP on special computer architectures, such as linear processor array, meshed processor graph, and partial *k*-tree communication graph [7-11]. Finally, metaheuristic algorithms like simulated annealing have been used to derive good enough approximate solutions within reasonable CPU time [12-13].

The current study belongs to the last domain of TAP research. Our primary objective is to provide a roadmap for better utilizing metaheuristic approaches and to incorporate multiple metaheuristics into one integrative framework for solving TAP more effectively and efficiently. Previous studies have shown that the success of using metaheuristic algorithms depends on a proper administration of *exploration* and *exploitation search* in order to escape from local optimality [14]. Moreover, specific execution strategies may cause the same metaheuristic algorithm to behave differently. It is also noted that different metaheuristics have varying computational performances for distinct applications and varied allowable execution time. Therefore, a hybrid approach incorporating multiple metaheuristics may yield better performance than a single approach [15]. It is advisable to conduct a thorough comparative study on these metaheuristics [16] so as to combine their favorable features into one comprehensive approach.

In this paper, we first employ two metaheuristics-genetic algorithm and reinforcement learning-to compare with simulated annealing for solving the TAP. Then, by examining their computational results, a hybrid algorithm combining the two alternative metaheuristics is devised to solve the TAP more effectively and efficiently. The remainder of this paper is organized as follows. Section 2 formulates the TAP to be addressed in this paper. In Section 3, two metaheuristics based on genetic algorithm and reinforcement learning are proposed individually. Section 4 then follows and presents the comparative simulation results, based on which a

*Address correspondence to this author at the Department of Information Management, National Chi Nan University, 303 University Rd., Puli, Nantou 545, Taiwan; Tel: +886-49-2910960; Fax: +886-49-2915205;
E-mail: pyyin@ncnu.edu.tw

1874-0618/09

hybrid strategy is devised to incorporate the two metaheuristics. Finally, Section 5 concludes this paper.

2. PROBLEM FORMULATION

In this study, we develop metaheuristic algorithms for TAP that exhibits the following problem features.

- Some communication cost between two tasks is incurred if there is a communication need between them and if they are executed on different processors.
- Each processor is capacitated with certain resources (such as processing power or memory size).
- A fixed initiating cost will be incurred if the processor is used by at least one task.
- Each task consumes some units of the resources of the processor on which it is executed.
- The objective is to minimize the total fixed and communication costs.

Assume there are r application tasks to be assigned to n processors in a distributed computing system. There are communication needs among some pairs of the tasks. Let c_{ij} be the communication cost between tasks i and j if they are executed on different processors and s_k be the fixed cost if processor k is assigned at least one task (i.e., is initiated). Task i consumes a_i units of the resources from its executing processor, and processor k has b_k units of resources. We define a task assignment $\boldsymbol{X}$ as a matrix of $r \times n$ binary variables x_{ik} ($1 \le i \le r$ and $1 \le k \le n$), where $x_{ik} = 1$ if task i is assigned to processor k, or 0 otherwise. Further, let $y_k = 1$ if processor k is assigned at least one task, and $y_k = 0$ if no task is assigned to it. Our objective is to minimize the sum of fixed cost for using processors and communication cost incurred by a task assignment that satisfies the resource constraints. The task assignment problem (TAP) can be formulated as the following 0-1 quadratic integer programming problem.

$$\text{Minimize } COST(\boldsymbol{X}) = \sum_{k=1}^{n} s_k y_k + \sum_{i=1}^{r-1} \sum_{j=i+1}^{r} c_{ij} \left(1 - \sum_{k=1}^{n} x_{ik} x_{jk} \right), \quad (1)$$

$$\text{subject to} \quad \sum_{k=1}^{n} x_{ik} = 1, \ \forall\, i = 1, 2, \ldots, r \quad (2)$$

$$\sum_{i=1}^{r} a_i x_{ik} \le b_k y_k \qquad \forall\, k = 1, 2, \ldots, n \quad (3)$$

$$\sum_{i=1}^{r} x_{ik} \le r y_k \qquad \forall\, k = 1, 2, \ldots, n \quad (4)$$

$$x_{ik} \in \{0,1\}, \ \ y_k \in \{0,1\} \qquad \forall\, i, k$$

The first and second terms in the objective function (1) represent the total fixed cost and communication cost, respectively, incurred by the assignment $\boldsymbol{X}$. Constraint (2) states that each task should be assigned to exactly one processor. Constraint (3) ensures that the resource capacity of each processor is greater than or equal to the total amount of resources used by its assigned tasks, and constraint (4) guarantees that a processor is used if it is assigned at least one task.

It has to be pointed out that the (TAP) formulation considered above is related to the assignment-type problem (ATP) which determines an assignment of some items (tasks) to some resources (processors) so as to optimize an objective function and satisfy side constraints [17-18]. What separates TAP from other ATP instances, especially the classic and general assignment problems, is the quadratic objective function. This quadratic functional form due to communication needs between tasks makes the problem more complex and worth further examination.

Since this (TAP) formulation is a 0-1 integer program with a quadratic objective function and it is computationally prohibitive, its transformations to linear programs have been developed where approximate solutions are reported within reasonable computational time [4-5]. On the other hand, an alternative for solving TAP efficiently is to use metaheuristics. In the next section, we will conduct a comparative study of several metaheuristic algorithms and devise a hybrid strategy to combine their favorable features.

3. TWO METAHEURISTIC ALGORITHMS FOR TAP

The searching methods using metaheuristics can be classified into two classes, *perturbation* methods and *constructive* methods. The perturbation methods start with a full specification of a solution and then iteratively move to a neighboring solution by perturbing part of the specification. Some typical methods of this sort are simulated annealing [19] and tabu search [20]. Genetic algorithm performs in a similar manner but it fosters a set of candidate solutions rather than a singleton [21]. On the other hand, the constructive methods start with a partial specification of a solution and then search for the next most probable segments to build a full specification. Usually a network for selecting the solution segments is trained and the experience regarding the merit of choosing a segment is converted to a transition probability pertaining to the corresponding edge. Some typical methods of this kind are artificial neural networks [22], ant colony optimization [23], and reinforcement learning [24].

In order to understand the relative strengths and weaknesses of different metaheuristics for solving the TAP, we adopt separate algorithms from each category, namely genetic algorithm from the perturbation group and reinforcement learning algorithm from the constructive group.

3.1. Genetic Algorithms

Genetic algorithms (GAs) are metaheuristic algorithms based on natural genetic systems [21]. Each candidate solution to the underlying problem is represented by a binary string, called a chromosome, using a problem-specific coding scheme. The merit of individual string is evaluated through a fitness function that properly fits the optimization goal of the problem [25]. GAs search for a near-optimal solution by fostering a population of strings using three genetic operators: selection, crossover, and mutation. The evolution

process is iterated until the fitness of strings can be hardly improved or until a pre-specified number of generations is reached. We next present the principal GA features for the TAP.

3.1.1. Coding Scheme and Fitness Function

Each assignment corresponding to a candidate solution of the TAP can be encoded into a string as

$$\boldsymbol{A} = \alpha_1 \alpha_2 \cdots \alpha_r, \tag{5}$$

where $\alpha_i \in [1, n]$ represents the index of the processor to which task i is assigned. Note that index character α_i can be encoded in binary with $\lceil \log_2 n \rceil$ bits. String $\boldsymbol{A}$ can be easily transformed to a corresponding assignment $\boldsymbol{X}$ (see Eq. (1)) of r tasks to n processors; however, this assignment may violate constraints (2)-(4). The fitness of string $\boldsymbol{A}$, in a sense, is inversely proportional to the sum of the incurred cost and the capacity-exceeded amount of resource requirement. Thus, we define the fitness function as

$$f(\boldsymbol{A}) = K - \big(COST(\boldsymbol{A}) + E(\boldsymbol{A})\big), \tag{6}$$

where K is a constant, $COST(\boldsymbol{A})$ is the total fixed and communication costs (see Eq. (1)) incurred by the assignment corresponding to $\boldsymbol{A}$, and $E(\boldsymbol{A})$ is a possible excess of resource requirement over the capacity determined by

$$E(\boldsymbol{A}) = \sum_{k=1}^{n} \left(\max\left(\sum_{i=1}^{r} a_i x_{ik} - b_k y_k,\ 0 \right) \right). \tag{7}$$

The larger the value of $E(\boldsymbol{A})$, the less feasible the assignment $\boldsymbol{A}$ under consideration. When $E(\boldsymbol{A}) = 0$, it indicates that there is no violation of the resource-capacity constraints. Essentially, $E(\boldsymbol{A})$ is used as a penalty function measuring the infeasibility of a candidate solution and is incorporated into the optimization goal of our problem.

3.1.2. Genetic Operators

A population of strings according to the coding scheme (5) is generated at random. This population then repeatedly evolves to subsequent populations using the following three genetic operators.

- *Selection*. This operator selects strings with high fitness values to form the next population mimicking the natural selection of the fittest [26]. Each string of the current population is selected with a probability proportional to its fitness, i.e., $Select_i = \mathrm{f}(A_i) \Big/ \sum_{j=1}^{P} \mathrm{f}(A_j)$, where $Select_i$ is the selection probability of string $\boldsymbol{A_i}$ and P is the population size.
- *Crossover*. This operator randomly selects pairs of strings within the current population. For each pair of parent strings, a crossover position is randomly determined. Two offspring strings are produced by swapping the bits to the right hand position of the crossover site of both parent strings. The crossover operator is conducted with a crossover probability p_c whose value is usually drawn from the range of [0.6, 1.0]. With this crossover operation, the solution space is explored by interchanging information between strings. Since the highly fit strings occupy a large proportion of the population, they are likely to experience more trials of crossover operations and the search is navigated toward "good" regions of the solution space.
- *Mutation*. Each gene of a string may undergo mutation with a low probability p_m. Note that we perform the mutation operator on character scale instead of binary bit. This operator substitutes a new value for a random character to be mutated. Mutation preserves sufficient diversity between strings in the population and prevents the undesirable premature convergence. It also guarantees a non-zero probability of the search for any feasible string.

3.2. Reinforcement Learning

The reinforcement learning approach addresses the issue of how a simple agent can learn a complicated task through a sequence of trial-and-error interactions with its environment [24]. The agent examines the current state of its environment and makes a decision of choosing an action to perform. The state of the environment is, therefore, triggered by the agent's action and changed to another state. The agent observes the new state and receives an immediate reward regarding the desirability about the state transition. The process is repeated and the agent learns an optimal policy of choosing an action in a given state that maximizes the expected sum of the cumulative rewards received over time.

The optimal assignment of the TAP can be learned by a reinforcement learning algorithm. Fig. (**1**) depicts the relationship between tasks and processors. The directed graph consists of $r + 2$ layers of nodes. The first layer and the last layer contain only one node, which represent the starting node and the sinking node, respectively. The remaining r layers represent the possible assignments of the r tasks. Each of these layers contains n nodes and each of the n nodes corresponds to the assignment of this task to a specific processor. A path emanating from the starting node and terminating at the sinking node represents one possible assignment of the r tasks. Next, we describe the features of the reinforcement learning algorithm for the TAP.

- The set of environment states, $S = \{s_0, s_f\} \cup \{s_{i,j}\}_{1 \le i \le r, 1 \le j \le n}$. Elements s_0 and s_f are the initial state and the final state corresponding to the starting node and the sinking node, respectively, and $s_{i,j}$ indicates the state that task i is assigned to processor j.
- The set of agent actions, $A = \{a_i\}_{1 \le i \le n}$. Selecting action a_i to perform means assigning the next task to processor i.
- The set of scalar rewards, R. The reward value is computed by the reward function discussed below.

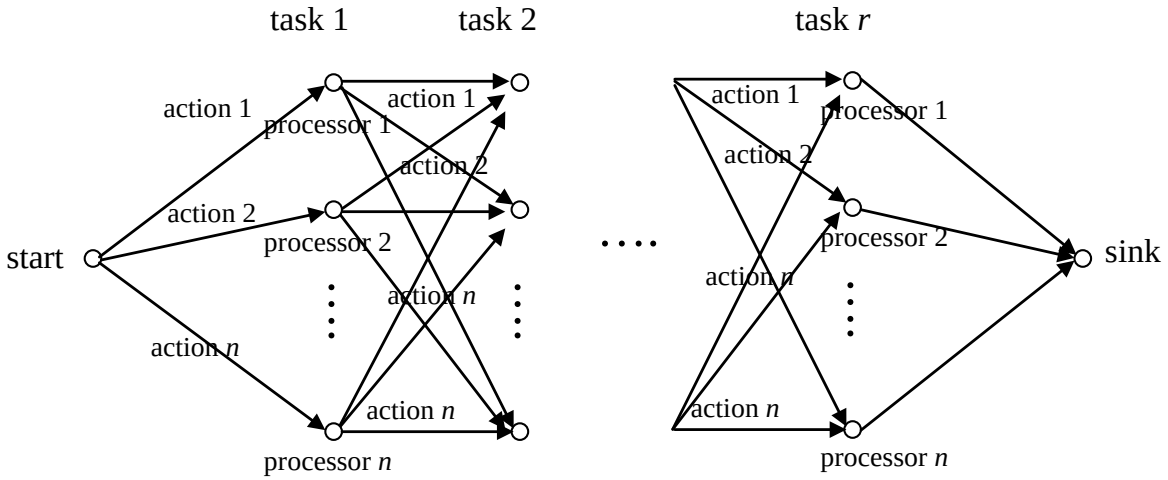

Fig. (1). The graph representation of the TAP as a reinforcement learning network.

- The state transition function, $\delta(s_{i,j}, a_k) = s_{i+1,k}$. This is apparent from the definitions of *S* and *A*.
- The reward function,

$$\rho(s_{i,j}, a_k) = \sum_{t=1}^{n}\left(b_t - \sum_{w=1}^{i+1} x_{wt} a_w\right) \Bigg/ \left(z_k + \sum_{w=1}^{i} c_{w,i+1}\left(1 - \sum_{t=1}^{n} x_{wt} x_{i+1,t}\right)\right) \quad (8)$$

- where $z_k = s_k$ if processor k is not used until the $(i+1)^{th}$ task is assigned to it; otherwise $z_k = 0$. As such, we measure the merit of assigning the $(i+1)^{th}$ task to processor k as the remaining resource capacity divided by the extra system costs incurred by the assignment of the $(i+1)^{th}$ task. We design the reward function to favor the assignment of the next task that both maximizes the remaining resource capacity (for later use) and incurs the least system cost (for optimization objective).

To learn the optimal assignment of the TAP, we employ the *Q*-learning algorithm [24], which has been one of the most commonly used methods for learning the optimal policy for reinforcement learning. First, we define the *Q* function, $Q(s_{i,j}, a_k)$, as the maximum cumulative reward which can be attained by performing action a_k in state $s_{i,j}$ and then proceeding optimally until the final state s_f is observed. The recursive definition of $Q(s_{i,j}, a_k)$ is given by

$$Q(s_{i,j},\ a_k) = \rho(s_{i,j},\ a_k) + \gamma \max_{a_l} Q(s_{i+1,k},\ a_l), \quad (9)$$

where $\gamma \in (0, 1)$ is the discounting factor that determines the relative value of the rewards received in the future. The agent then initializes a table of the estimate, $\hat{Q}(s_{i,j}, a_k)$, of the *Q* function for each possible state-action pair $(s_{i,j}, a_k)$. The initial value of $\hat{Q}(s_{i,j}, a_k)$ can be any small constant. For simplicity, we initialize each entry of $\hat{Q}(s_{i,j}, a_k)$ as 1. These table entries are iteratively updated by

$$\hat{Q}(s_{i,j},\ a_k) = \rho(s_{i,j},\ a_k) + \gamma \max_{a_l} \hat{Q}(s_{i+1,k},\ a_l) \quad (10)$$

In other words, the value of the generic $Q(s_{i,j}, a_k)$ can be incrementally approximated by $\hat{Q}(s_{i,j}, a_k)$. The *Q*-learning algorithm for the TAP is summarized in Table **1**. The algorithm is repeated for a pre-specified maximum number of iterations. Then the *r* sequential actions chosen by the learned optimal policy constitute the near-optimal assignment. There still remains an issue of how to choose the action in a given state (see Step 4 in Table **1**). Obviously, if every action can be visited infinitely often, the policy learned will converge to the optimal task assignment. However, in real-world applications where the computation time is limited, it is crucial to design an appropriate action selection rule which instructs the agent to experience the minimal number of actions that still explore the policy space sufficiently.

Let the agent be in state $s_{i,j}$ and face the choice among a set of available actions $\{a_k\}_{1 \le k \le n}$. We propose a thresholded maximum-exploitation action selection rule to determine the probability of choosing action a_k as follows.

$$p(a_k \mid s_{i,j}) = \begin{cases} 1 & \text{, if } q < q_0 \text{ and } k = \arg\max_l \hat{Q}(s_{i,j}, a_l); \\ 0 & \text{, if } q < q_0 \text{ and } k \neq \arg\max_l \hat{Q}(s_{i,j}, a_l); \\ \dfrac{1}{n} & \text{, otherwise,} \end{cases} \quad (11)$$

where q is a random number drawn from U(0, 1), and $q_0 \in (0, 1)$ is the threshold controlling the relative emphasis on each sub-rule. The tie with $p(a_k \mid s_{i,j})$ is broken at random. This rule facilitates the controlled tradeoff between the selection of the action that delivers the maximum estimate of $\hat{Q}$ and a uniform random selection. It has been empirically shown that the thresholded maximum-exploitation action selection rule outperforms several other competing rules [27].

Table 1. The *Q*-Learning Algorithm for the TAP

Step 1 Initialize the table entry $\hat{Q}(s_{i,j}, a_k) = 1$ for each *i*, *j*, *k*.
Step 2 Set *Iteration* = 1.
Step 3 Start with the initial state s_0.
Step 4 Select an action according to the action selection rule.
Step 5 Update the table entry $\hat{Q}(s_{i,j}, a_k)$ using Eq. (10).
Step 6 If the final state s_f is not yet reached, goto Step 4.
Step 7 If *Iteration* < *MAX_ITERATION* //stopping criterion is not yet satisfied// Set *Iteration* = *Iteration* + 1, and goto Step 3.
Step 8 Start from the initial state s_0, output the sequence of actions which result in the maximum $\hat{Q}$ values until the final state s_f is reached.

4. SIMULATION RESULTS

In this section, we present the performance evaluations obtained using the competing metaheuristics. Three algorithms, namely simulated annealing (SA), genetic algorithm (GA), and *Q*-learning algorithm (QA), were implemented in C++ language and executed on a 1.8GHz PC with 192MB RAM. The parameters used in these algorithms are tuned empirically. To be specific, the parameter setting used by SA is (initial temperature = 500, length of Markov chain = 1200, temperature decreasing rate = 0.85, chain increasing rate = 1.1), GA (population size = 100, crossover rate = 0.9, mutation rate = 0.03), and QA (action selection threshold = 0.3).

The inter-task communication is rendered by a task interaction graph (TIG), $G(V,E)$, where V is a set of r nodes and E is a set of edges connecting these nodes. Each node represents a task and each edge specifies the communication requirement between its two connecting tasks. Fig. (**2**) gives an example of TIG where the communication requirement among four tasks is specified. We define the task interaction density d of $G(V,E)$ as

$$d = \frac{|E|}{|V|(|V|-1)/2}, \tag{12}$$

where $|V|$ and $|E|$ denote the number of elements in the corresponding set.

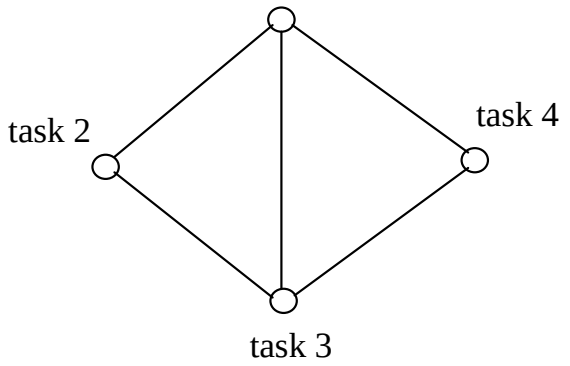

Fig. (2). An example of TIG.

The testing dataset is generated according to different problem characteristics, namely the r/n ratio and density d. We set the value of n equivalent to 6, 10, 20, and 30, respectively, and compute the value of r accordingly so that the r/n ratio is equal to 1.5, 2, and 3. For each (r, n) pair, we generate three different TIGs with density d equivalent to 0.3, 0.5, and 0.8. As such, we obtain a testing dataset of 36 problem instances available from [28] for evaluating the performance of the competing methods. The other parameters are generated randomly: the fixed cost is between 1 and 200, the communication cost is between 1 and 50, the resource capacity of each processor varies from 50 to 250, and the resource requirement of each task ranges from 1 to 50.

We analyze both off-line and on-line performances. The off-line performance is measured as the average cost obtained when the testing algorithm has run for a specified period of CPU time. This is a common practice that many previous studies have adopted. However, this measure alone can be inadequate for analyzing the strengths and weaknesses of metaheuristic algorithms because different algorithms may exhibit different performance levels as CPU duration time varies. Thus, we also measure the on-line performance as the dynamic costs to allow for varied CPU elapse time.

4.1. Off-Line Performance

We conduct two experiments as follows. In the first experiment, all algorithms are given equally short CPU time periods to derive the solutions, while in the second experiment the allowed CPU time durations are relatively longer. As for the short CPU time periods, they are 10, 50, 100, and 200 seconds for various numbers of processors. And for the long CPU time periods, they are 100, 500, 1000, and 2000 seconds. Since all algorithms are probabilistic and each independent run of the same algorithm on a particular testing problem may yield a different result, we calculate the average cost over 10 independent runs of each algorithm for every problem instance.

- Performance with Short CPU Elapse Time

Table **2** tabulates the average costs obtained using the three metaheuristic algorithms given the short CPU time

Table 2. The Average Costs and Offsets Obtained Using the Three Metaheuristics, Namely the Simulated Annealing (SA), Genetic Algorithm (GA), and *Q*-Learning Algorithm (QA), Given a Short CPU Elapse Time

r	n	d	SA		GA		QA		CPU Time
			Cost	Offset%	Cost	Offset%	Cost	Offset%	
9	6	0.3	**372**	0.00%	**372**	0.00%	**372**	0.00%	
		0.5	**482**	0.00%	**482**	0.00%	**482**	0.00%	
		0.8	**419**	0.00%	**419**	0.00%	**419**	0.00%	
12	6	0.3	**805**	0.00%	**805**	0.00%	**805**	0.00%	
		0.5	**968**	0.00%	970	0.21%	**968**	0.00%	10 seconds
		0.8	995	0.60%	1004	1.49%	**989**	0.00%	
18	6	0.3	**1257**	0.00%	**1257**	0.00%	**1257**	0.00%	
		0.5	2256	1.15%	2258	1.24%	**2230**	0.00%	
		0.8	3032	0.59%	3016	0.07%	**3014**	0.00%	
15	10	0.3	970	1.55%	966	1.14%	**955**	0.00%	
		0.5	1546	4.72%	1528	3.60%	**1473**	0.00%	
		0.8	**1961**	0.00%	1973	0.61%	**1961**	0.00%	
20	10	0.3	1785	1.23%	1878	6.12%	**1763**	0.00%	
		0.5	2777	1.69%	2797	2.40%	**2730**	0.00%	50 seconds
		0.8	4927	2.07%	4853	0.58%	**4825**	0.00%	
30	10	0.3	4377	5.32%	4283	3.25%	**4144**	0.00%	
		0.5	7382	1.37%	7332	0.70%	**7281**	0.00%	
		0.8	14658	3.87%	14131	0.28%	**14091**	0.00%	
30	20	0.3	4972	2.84%	4908	1.57%	**4831**	0.00%	
		0.5	8392	1.63%	8312	0.69%	**8255**	0.00%	
		0.8	12332	1.79%	12269	1.29%	**12111**	0.00%	
40	20	0.3	8633	1.97%	8548	0.99%	**8463**	0.00%	
		0.5	14817	2.44%	14629	1.19%	**14455**	0.00%	100 seconds
		0.8	**23607**	0.00%	23824	0.91%	**23608**	0.00%	
60	20	0.3	**20533**	0.00%	20686	0.74%	20664	0.63%	
		0.5	**33818**	0.00%	34357	1.57%	34108	0.85%	
		0.8	58651	2.13%	57799	0.69%	**57403**	0.00%	
45	30	0.3	11431	2.18%	11223	0.37%	**11182**	0.00%	
		0.5	21358	0.58%	21412	0.83%	**21235**	0.00%	
		0.8	**30914**	0.00%	31619	2.23%	31433	1.65%	
60	30	0.3	20073	0.77%	19968	0.25%	**19919**	0.00%	
		0.5	**37301**	0.00%	37800	1.32%	37443	0.38%	200 seconds
		0.8	**59508**	0.00%	60067	0.93%	59958	0.75%	
90	30	0.3	**50975**	0.00%	51312	0.66%	51010	0.07%	
		0.5	89305	1.27%	89629	1.63%	**88169**	0.00%	
		0.8	146589	1.42%	144928	0.29%	**144503**	0.00%	
Average	offset			1.20%		1.11%		0.12%	

periods. It is observed that QA is better than or equal to the other two algorithms by obtaining the minimal cost for 30 instances. For the other six instances, SA produces the minimal cost. To compare the results with absolute per-

Table 3. The Minimum Costs Obtained Using the Lingo Package and the Used CPU Time

r	n	d	Minimum Cost	CPU Time
9	6	0.3	372	36 sec
		0.5	471	1 min 14 sec
		0.8	399	1 min 37 sec
12	6	0.3	769	5 min 16 sec
		0.5	925	18 min 52 sec
		0.8	954	23 min 18 sec
18	6	0.3	1164	30 min 56 sec
		0.5	2220	7 hr 22 min 56 sec
		0.8	2859	8 hr 38 min 10 sec
15	10	0.3	941	1 hr 52 min 32 sec
		0.5	1414	22 hr 15 min 53 sec
	others		unknown	1 day

formance, we adopt [4] to transform the quadratic objective function (1) into an integer linear program and use the Lingo package to compute the exact solution. The maximum allowed CPU time for solving an instance by Lingo is set to one day. Table **3** presents the derived minimum costs and the CPU time used by the package. When Lingo fails to solve the integer linear program within one day, it is terminated with infeasible solution or unknown status and we discard such cases for further comparison. It can be seen that as the problem size increases, it becomes too time consuming for Lingo to compute exact solutions due to the enormous computations required. For those testing instances where exact solutions are available, QA is shown to have much closer approximate results, while SA and GA deviate significantly from the exact solutions.

To provide a clearer view on the comparative performances among the competing methods, the cost offset (defined as the difference to the minimal cost of the three algorithms divided by the corresponding cost) is also calculated and listed in Table **2**. We observe that the cost offset ranges from 0% to 5.32% for SA, 0% to 6.12% for GA, and 0% to 1.65% for QA. The average cost offsets over all the instances are 1.20%, 1.11%, and 0.12% for SA, GA, and QA, respectively.

- Performance with Long CPU Elapse Time

The experimental result with long CPU elapse time is displayed in Table **4**. We observe that GA attains the minimal cost in all cases, QA reports moderate results, and SA exhibits the worst performance. The offset to the minimal cost of GA varies from 0% to 2.64% for QA, and 0% to 4.94% for SA. The average cost offset is 0.51% for QA and 1.66% for SA.

We observe from the above experiments that the comparative performances between the three metaheuristics with long CPU elapse time are inconsistent with those with short CPU time duration. Hence, we cannot conclude that any particular algorithm is superior to the others. A more sophisticated performance evaluation is thus required.

4.2. On-Line Performance

Since the performance levels of the competing algorithms vary as the CPU elapse time increases, we also analyze the on-line performance which is the cost offset with respect to the varied lengths of CPU time duration. Fig. (**3**) illustrates a typical run in our experiment. It is observed that QA has the minimal cost offset when the length of CPU time duration is short (less than 668 seconds) in this particular run, while GA replaces QA as the best approach when the CPU duration becomes long enough. However, the time point at which GA and QA exchange their roles as the best approach is case-dependent, i.e., for some instances, GA may converge faster to the minimal cost, but for others GA would need more explorative search before it generates a better solution. SA, however, is consistently the least effective among the three; its cost offset ranges from 0.97% to 3.49%.

4.3. A Hybrid Strategy

It has been successfully shown in many previous studies that a hybrid version combining multiple metaheuristics can improve the performance over a single approach [15, 29-30]. It is also advised that exploration search should be emphasized in the early stage of the evolution for finding good regions in the solution space. As the evolution proceeds, exploitation search can be focused to further improve the candidate solutions. From the experimental results given in previous sections, it manifests that GA has focused on exploration search and can deliver high quality solutions if enough CPU time is used, while QA intensifies the exploitation search and usually gets to a good solution consuming only a short CPU time period. We thus devise a hybrid method (HYB) to incorporate GA and QA, which are the best approaches based on our empirical results presented in previous sections. The hybrid method consists of two main stages. In the *first* stage, GA is applied for exploring the solution space until it cannot further improve the solution for a period of time that is equivalent to one fifth of the total allowed CPU elapse time. Upon the termination of GA, all candidate

Table 4. The Average Costs and Offsets Obtained Using the Three Metaheuristics, Namely the Simulated Annealing (SA), Genetic Algorithm (GA), and *Q*-Learning Algorithm (QA), Given a Long CPU Elapse Time

r	n	d	SA		GA		QA		CPU Time
			Cost	Offset%	Cost	Offset%	Cost	Offset%	
9	6	0.3	**372**	0.00%	**372**	0.00%	**372**	0.00%	
		0.5	**482**	0.00%	**482**	0.00%	**482**	0.00%	
		0.8	**419**	0.00%	**419**	0.00%	**419**	0.00%	
12	6	0.3	**805**	0.00%	**805**	0.00%	**805**	0.00%	
		0.5	969	0.10%	**968**	0.00%	**968**	0.00%	100 seconds
		0.8	**989**	0.00%	**989**	0.00%	**989**	0.00%	
18	6	0.3	**1257**	0.00%	**1257**	0.00%	**1257**	0.00%	
		0.5	**2230**	0.00%	**2230**	0.00%	**2230**	0.00%	
		0.8	**2992**	0.00%	**2992**	0.00%	**2992**	0.00%	
15	10	0.3	957	0.21%	**955**	0.00%	**955**	0.00%	
		0.5	1494	1.41%	**1473**	0.00%	**1473**	0.00%	
		0.8	1969	0.41%	**1961**	0.00%	**1961**	0.00%	
20	10	0.3	1768	0.28%	**1763**	0.00%	**1763**	0.00%	
		0.5	2772	1.52%	**2730**	0.00%	**2730**	0.00%	500 seconds
		0.8	4926	2.05%	**4825**	0.00%	**4825**	0.00%	
30	10	0.3	4262	2.86%	**4140**	0.00%	**4140**	0.00%	
		0.5	7419	3.61%	**7151**	0.00%	7225	1.02%	
		0.8	14509	4.45%	**13864**	0.00%	13958	0.67%	
30	20	0.3	4995	3.82%	**4804**	0.00%	**4804**	0.00%	
		0.5	8424	3.16%	**8158**	0.00%	8180	0.27%	
		0.8	12426	4.94%	**11812**	0.00%	11973	1.34%	
40	20	0.3	8545	2.19%	**8358**	0.00%	8362	0.05%	
		0.5	14582	4.31%	**13953**	0.00%	14248	2.07%	1000 seconds
		0.8	23628	2.35%	**23073**	0.00%	23454	1.62%	
60	20	0.3	20467	1.81%	**20096**	0.00%	20559	2.25%	
		0.5	33980	1.97%	**33309**	0.00%	33820	1.51%	
		0.8	58806	2.91%	**57092**	0.00%	57227	0.24%	
45	30	0.3	11290	2.33%	**11027**	0.00%	11068	0.37%	
		0.5	21277	1.29%	**21003**	0.00%	21137	0.63%	
		0.8	31340	3.32%	**30299**	0.00%	31120	2.64%	
60	30	0.3	19951	1.12%	**19727**	0.00%	19790	0.32%	
		0.5	37224	0.96%	**36868**	0.00%	37165	0.80%	2000 seconds
		0.8	59334	1.88%	**58217**	0.00%	58671	0.77%	
90	30	0.3	50964	1.55%	**50173**	0.00%	50991	1.60%	
		0.5	89202	1.30%	**88044**	0.00%	88081	0.04%	
		0.8	146430	1.65%	**144017**	0.00%	144072	0.04%	
Average	offset			1.66%		0.00%		0.51%	

solutions in the last population and the best solution obtained up to that point are sorted in the increasing order of their costs. The top 5% of the sorted solutions are then used to train the *Q*-learning network to derive the initial *Q* value

estimates. The training process is also performed using the *Q*-learning algorithm presented in Table **1** except that in Step 4 the algorithm assigns the next task according to the training solution that is fed into the network. In the *second* stage, QA is applied using the remaining CPU elapse time to report the final best solution when it terminates. The proposed hybrid method takes advantage of the exploration power of GA to derive better initial *Q* value estimates, and provides diversification of candidate global best solutions to prevent getting trapped in local minima. Then the intensification power of QA is used to guide the search in the candidate solution areas found by GA and derive the best solution quickly.

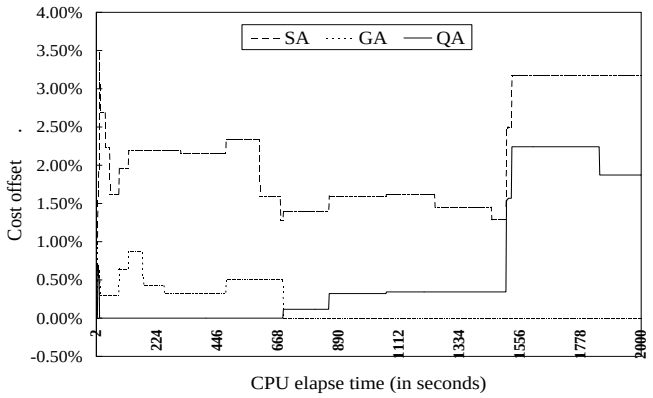

Fig. (3). The cost offsets in the variance of CPU elapse time.

- Off-Line Performance

Table **5** shows the costs obtained using HYB given short and long CPU time, respectively, and the corresponding offsets to the minimal cost obtained using the three original metaheuristics. It is found that the proposed HYB method either outperforms or performs equally well as the three metaheuristic algorithms for all problem instances. Depending on the problem complexity, the cost offset to the minimal cost of the three metaheuristics varies from 0.0% to –1.33% for short CPU time and 0.0% to –0.97% for long CPU time, respectively. Note that a negative value of cost offset indicates the derived cost by HYB is smaller. The average offset is –0.31% if the short CPU time periods are given. For the cases with long CPU time periods, HYB obtains an average offset of –0.18%.

- On-Line Performance

Fig. (**4**) shows a typical run of the cost variations with short CPU elapse time for all testing methods. We observe that QA intensifies the search toward the neighborhood of the best-so-far solution and the cost drops quickly at the early stage. However, it is hardly improved after this period. On the other hand, GA provides diversified initialization and may improve the solution if the CPU elapse time is long enough. The cost obtained by SA is also improved with the increment of CPU elapse time, but at a slower rate than GA. The proposed HYB method first applies GA and thus behaves similarly like GA within this period. Then HYB uses the top 5% candidate solutions in the last GA population and the best-so-far solution to train the *Q*-learning network and switches to the QA process. The solution of HYB is improved dramatically after this transition and finally yields a much better result because QA now can derive a better solution by exploiting the knowledge learned from GA.

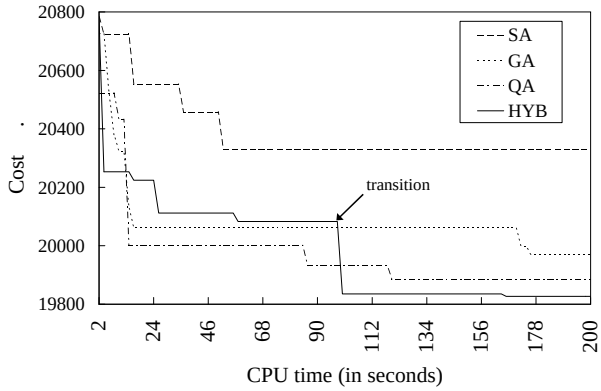

Fig. (4). The cost variations with short CPU elapse time.

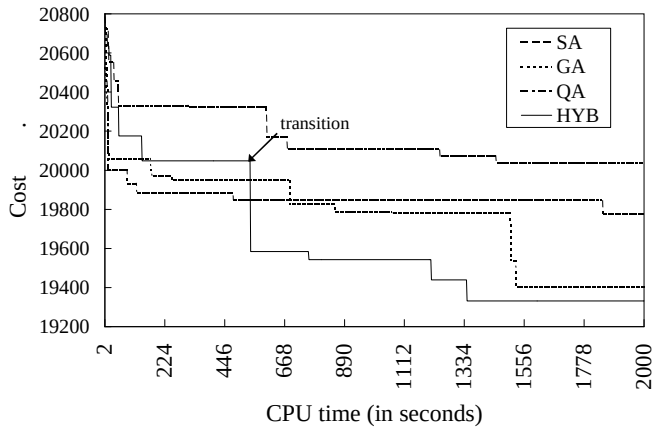

Fig. (5). The cost variations with long CPU elapse time.

Fig. (**5**) illustrates the results corresponding to the experiment with long CPU elapse time. We observe that GA outperforms SA and QA if the allowed CPU elapse time is long enough. However, the proposed HYB method, which takes advantage of GA and QA, can still derive the best solution among all.

- *t*-Test Analysis

To further confirm the relative performances of the various algorithms, we conduct the matched pair *t*-test for significance on cost difference. Table **6** displays the *t*-test on the cost difference between every pair of two different algorithms for the 36 testing instances. In summary, the order of these algorithms from the best to the worst is HYB, QA, GA, and SA if the short CPU time is specified, while the order changes to HYB, GA, QA, and SA for the case of long CPU time. Since the confidence coefficient is 1.689 for the 95% confidence interval over 36 samples, we also observe that all of the cost differences are statistically significant except for the one case of SA *vs* GA for the short CPU elapse time. Since the allowed CPU elapse time for deriving the optimal solution can vary significantly in different applications, it is beneficial to consider HYB as the candidate approach for it guarantees the best performance among these metaheuristics with a wide range of allowed CPU elapse time.

Table 5. The Average Costs Obtained Using the Hybrid Method (HYB) with their Offsets to the Minimal Costs from Tables 2 and 4

r	n	d	Cost with Short CPU Time	Offset to Minimal Cost in Table 2	Cost with Long CPU Time	Offset to Minimal Cost in Table 4
9	6	0.3	372	0.00%	372	0.00%
		0.5	482	0.00%	482	0.00%
		0.8	419	0.00%	419	0.00%
12	6	0.3	805	0.00%	805	0.00%
		0.5	968	0.00%	968	0.00%
		0.8	989	0.00%	989	0.00%
18	6	0.3	1257	0.00%	1257	0.00%
		0.5	2230	0.00%	2230	0.00%
		0.8	2992	-0.74%	2992	0.00%
15	10	0.3	955	0.00%	955	0.00%
		0.5	1473	0.00%	1473	0.00%
		0.8	1961	0.00%	1961	0.00%
20	10	0.3	1763	0.00%	1763	0.00%
		0.5	2730	0.00%	2730	0.00%
		0.8	4825	0.00%	4825	0.00%
30	10	0.3	4140	-0.10%	4140	0.00%
		0.5	7229	-0.72%	7151	0.00%
		0.8	13950	-1.01%	13819	-0.33%
30	20	0.3	4804	-0.56%	4804	0.00%
		0.5	8214	-0.50%	8153	-0.06%
		0.8	12049	-0.51%	11791	-0.18%
40	20	0.3	8401	-0.74%	8341	-0.20%
		0.5	14265	-1.33%	13842	-0.80%
		0.8	23308	-1.28%	22977	-0.42%
60	20	0.3	20291	-1.19%	19994	-0.51%
		0.5	33772	-0.14%	33292	-0.05%
		0.8	57376	-0.05%	56779	-0.55%
45	30	0.3	11087	-0.86%	10931	-0.88%
		0.5	21217	-0.08%	20981	-0.10%
		0.8	30833	-0.26%	30272	-0.09%
60	30	0.3	19862	-0.29%	19537	-0.97%
		0.5	37244	-0.15%	36782	-0.23%
		0.8	59379	-0.22%	58125	-0.16%
90	30	0.3	50855	-0.24%	49801	-0.75%
		0.5	88144	-0.03%	87983	-0.07%
		0.8	144228	-0.19%	143943	-0.05%
Average	offset			-0.31%		-0.18%

Table 6 The Matched Pair *t*-Test for Significance of Cost Difference Between Every Pair of Two Different Algorithms

Short CPU Duration	**GA**	**QA**	**HYB**
SA	-0.236	-2.020*	-3.326*
GA		-3.339*	-4.608*
QA			-3.801*
Long CPU Duration	**GA**	**QA**	**HYB**
SA	-4.333*	-3.121*	-4.462*
GA		3.596*	-3.390*
QA			-3.928*

*Statistically significant at the .05 level.

5. CONCLUSIONS

In many problem domains, we are required to assign the tasks of an application to a set of distributed processors such that system costs are minimized. Several versions of the task assignment problem (TAP) have been formally defined but, unfortunately, most of them have been known to be NP-complete. To our knowledge, there is little research discussing the comparative performances for solving TAP using different metaheuristics. In this paper, we have implemented simulated annealing (SA) algorithm, genetic algorithm (GA), and *Q*-learning algorithm (QA) to help solve the TAP. The computational experience manifests that QA outperforms SA and GA when a short CPU elapse time is allowed, and GA turns out to be the best approach for long CPU time duration. Suggested by the empirical results, we proceed to devise a hybrid method (HYB) which first applies GA to explore potentially good areas in the solution space and uses the evolved quality solutions to train the *Q*-learning network. QA is then applied to intensify the search and find the final best solution. Experimental results show that HYB can derive the best quality solutions among all the testing metaheuristic algorithms for a wide variation of CPU elapse time, and the cost difference to the other approaches is statistically significant.

REFERENCES

[1] D. Ghosh, I. Murthy, and A. Moffett, "File allocation problem: comparisons of models with worst case and average communication delays", *Operations Research*, vol. 40, pp. 1074-1085, 1992.

[2] Z. Liu, and R. Righter, "Optimal load balancing on distributed homogeneous unreliable processors", *Operations Research*, vol. 46, pp. 563-573, 1998.

[3] V. M. Lo, "Task assignment in distributed systems", Ph.D. thesis, University of Illinois, 1983.

[4] A. Ernst, H. Hiang, and M. Krishnamoorthy, "Mathematical programming approaches for solving task allocation problems", in 16th National Conf. of Australian Society of Operations Research, 2001.

[5] A. Billionnet, M. C. Costa, and A. Sutter, "An efficient algorithm for a task allocation problem", *Journal of ACM*, vol. 39, pp. 502-518, 1992.

[6] G. H. Chen, J. S. Yur, "A branch-and-bound-with-underestimates algorithm for the task assignment problem with precedence constraint", in 10th International Conf. on Distributed Computing Systems, 1990, pp. 494-501.

[7] V. M. Lo, "Heuristic algorithms for task assignment in distributed systems", *IEEE Trans Computers*, vol. 37, pp. 1384-1397, 1988.

[8] D. M. Nicol, and D. R. O'Hallaron, "Improved algorithm for TAP pipelined and parallel computations", *IEEE Trans Computers*, vol. 40, pp. 295-306, 1991.

[9] D. Fernandez-Baca and A. Medepalli, "Parametric task allocation on partial k-trees", *IEEE Trans Computers*, vol. 46, pp. 738-742, 1993.

[10] C. H. Lee, and K.G. Shin, "Optimal task assignment in homogeneous networks", *IEEE Trans Parallel and Distributed Systems*, vol. 8, pp. 119-129, 1997.

[11] M. Kafil, and I. Ahmad, "Optimal task assignment in heterogeneous distributed computing systems", *IEEE Concurrency*, vol. 6, pp. 42-50, 1998.

[12] F. T. Lin and C. C. Hsu, "Task assignment scheduling by simulated annealing", in *Conference on Computer and Communication Systems* 279-283, 1990, pp. 279-283.

[13] Y. Hamam, and K. S. Hindi, "Assignment of program tasks to processors: A simulated annealing approach", *European Journal of Operational Research*, vol. 122, pp. 509-513, 2000.

[14] Z. Michalewicz and D. B. Fogel, *How to Solve It: Modern Heuristics*. Springer-Verlag, New York, 2002.

[15] Z. Z. Lin, J. C. Bean, and C. C. White, "A hybrid genetic/optimization algorithm for finite-horizon, partially observed Markov decision processes", *INFORMS Journal on Computing*, vol. 16, pp. 27-38, 2004.

[16] M. C. Fu, "Optimization for simulation: theory *vs* practice", *INFORMS Journal on Computing*, vol. 14, pp. 192-215, 2002.

[17] J. A. Ferland, A. Hertz, and A. Lavoie, "An object-oriented methodology for solving assignment-type problems with neighborhood search techniques", *Operations Research*, vol. 44, pp. 347-359, 1996.

[18] B. B. M. Shao, and H. R. Rao, "A comparative analysis of information acquisition mechanisms for discrete resource allocation", *IEEE Trans Systems Man Cybernetics A*, vol. 31, pp. 199-209, 2001.

[19] S. Kirkpatrick, C. Gelatt Jr., and M. Vecchi, "Optimization by simulated annealing", *Science*, vol. 220, pp. 671-680, 1983.

[20] F. Glover, "Tabu search – Part I", *ORSA J. Computing*, vol. 1, pp. 190-206, 1989.

[21] D. E. Goldberg, *Genetic Algorithms: Search, Optimization and Machine Learning*, Addison-Wesley, Reading, MA, 1989.

[22] R. Hecht-Nielsen, *Neurocomputing*, Addison-Wesley, Reading, MA, 1990.

[23] M. Dorigo, and L. Gambardella, "Ant colony system: a cooperative learning approach to the traveling salesman problem", *IEEE Trans Evolutionary Computation*, vol. 1, pp. 53-66, 1997.

[24] L. P. Kaelbling, and A. W. Moore, "Reinforcement learning: a survey", *Journal of Artificial Intelligence Research*, vol. 4, pp. 237-285, 1996.

[25] H. C. Tang, and C. Kao, "Searching for good multiple recursive random number generators *via* a genetic algorithm", *INFORMS Journal on Computing*, vol. 16, pp. 284-290, 2004.

[26] C. C. Aggarwal, J. B. Orlin, and R. P. Tai, "Optimized crossover for the independent set problem", *Operations Research*, vol. 45, pp. 226-234, 1997.

[27] P. Y. Yin, "Maximum entropy-based optimal threshold selection using deterministic reinforcement learning with controlled randomization", *Signal Processing*, vol. 82, pp. 993-1006, 2002.

[28] Experimental dataset downloadable from http://www.im.ncnu.edu.tw/~pyyin/public_html/MAP_problems.zip

[29] A. H. Mantawy, Y. L. Abdel-Magid, and S. Z. Selim, "Integrating genetic algorithms, tabu search, and simulated annealing for the unit commitment problem", *IEEE Trans Power Systems*, vol. 14, pp. 829-836, 1999.

[30] C. F. Liaw, "A hybrid genetic algorithm for the open shop scheduling problem", *European Journal of Operational Research*, vol. 124, pp. 28-42, 2000.

Received: November 26, 2008 Revised: January 21, 2009 Accepted: January 26, 2009