Memetic algorithm approach for multi-criteria network scheduling

by

Jarosław Rudy, Dominik Żelazny
Wrocław University of Technology,
Institute of Computer Engineering, Control and Robotics
Wybrzeże Wyspiańskiego 27, 50-360, Wrocław
{jaroslaw.rudy,dominik.zelazny}@pwr.wroc.pl
http://www.iiar.pwr.wroc.pl

Research paper

ABSTRACT

In era of Internet-based services and cloud computing the nature of a Web services is evolving. Instead of resource-retrieval that takes few miliseconds, the clients' requests may contain considerable amount of data and require minutes or even hours to perform all necessary computations and produce a proper response. This is also true in case of server farms or computer clusters that use single dispatcher or load-balancer to control the access to back-end servers. However with such high computation time and increase in clients' requirements regarding Quality of Service it is possible to use more sophisticated methods of dispatching the clients' requests and meeting various service criteria. We use an online version of enhanced Local Search Elitist Non-dominated Sorting Genetic Algorithm (or LSNSGA-II) for multi-criteria scheduling of incoming requests and compare it with other approaches to load-balancing including Round Robin, Shortest Job First and Earliest Deadline First implementations.

Keywords:

Multi-criteria network scheduling, load balancing, memetic algorithms, Pareto efficiency, quality of service.

INTRODUCTION

In the era of Internet-based services and cloud computing more and more tasks that previously had to be performed on a local machine can now be done remotely. The users of such services no longer require any specialized equipment or software on their own. Client software and network connection is all that is needed to access particular service and perform required task or computation on a remote server. Moreover, a single service is often implemented by a set of identical back-end servers. In order to manage multiple requests certain algorithms are used to assign requests to specific back-end servers. In case of classic requests (*e.g.* HTTP site request) the time is crucial, so simple and fast algorithms (like Round Robin, for example) are used to choose back-end machine that will fulfill a given request. Those approaches perform load balancing by distributing the requests equally among available servers and by meeting Quality of Service, most commonly understood in terms of computer network conditions, like latency time or throughput.

In this paper we tried a different approach. We consider special class of requests and services, where time needed to produce a response is long enough, that more sophisticated and time-consuming methods – like genetic or memetic algorithms – can be used to schedule incoming requests. We also consider Quality of Service on a higher level: both service users and service providers may specify several criteria (like response time for the users, completion time for the providers and tardiness for both) they would prefer to optimize. Therefore we use a memtic algorithm for multi-criteria network scheduling and compare the results against few fast and simple constructive algorithms. The remainder of this paper is organized as follows. In Section 2 we present short overwiev of the problem of network scheduling and load-balancing in its classic meaning and then, in Section 3, we focus on our approach to mutli-criteria network scheduling of time-consuming tasks. In section 4 we describe our memetic algorithm used for network scheduling and its background. Section 5 contains results of comparison research of our algorithm and several constructive algorithms. Finally, Section 6 presents the conclusions and the paper's summary.

CLASSIC NETWORK SCHEDULING

In classic network scheduling single Internet service is most commonly implemented as a set of back-end servers that are either identical or differ in terms of performance (*i.e.* the time needed to produce response for a given request), typically in the form of a server farm. Commonly, requests are relatively short and simple and can be handled by any of the back-end servers. In this case the requests are dispatched as soon as they arrive in the system and fast algorithms like Least Busy Machine or Round Robin are used to divide the work equally amongst the servers, thus implementing basic load-balancing. This allows to reduce the number of overloaded or idle machines (and, in result, the power usage), the average time needed to produce a response and balance the transfer in computer network itself. This, in turn, is observed by the client and determines whether certain Quality of Service (QoS) conditions have been met. Unfortunately, the desired level of QoS is rarely specified by the client and mostly remains as a low-level parameter of the computer network connection. However other quality factors, like Quality of Experience or Quality of Service exist and can be used to measure quality on a higher level.

When discussing the problems of request dispatching and load-balancing it is important to consider the place in the system where above tasks will be performed. Cardellini *et al.* [1] distinguish four different types of request dispatching:

1. Client-based. In this approach clients themselves are able to choose the server they want to send requests to. It also includes proxy servers, which can store past responses. This approach has limited applications and lacks scalability.
2. DNS-based. This situation typically involves mapping single URL to several different Internet Addresses (IPs), each on different machine through the use of Domain Name Service. This approach is a low-level solution, however the back-end servers remain transparent to the outside world and the users.
3. Dispatcher-based. In this case all client-server routing is managed by single centralized network component. It allows much higher level of control, but can lead to performance problems, when network packets are overwritten with the address of the actual back-end server. Decentralization may also cause a bottleneck, since the dispatcher is the only component with knowledge about back-end servers and all requests must pass through it.
4. Server-side. In this approach servers themsleves have the ability to dispatch tasks and perform load-balancing. Initially, the request is send to one of the back-end servers by the use of one of the previous approaches. The difference is: the chosen server may decide to redirect the task to another server. This means that the server-side approach is an example of a distributed system, where back-end servers have knowledge about other servers and may cooperate to solve tasks.

The first and second approach lack in scalability or control, so we won't consider them any further in this paper. The fourth approach requires more consideration. Distributed systems, where the tasks may migrate beetwen different nodes are quite common and perform tasks dispatching in various ways. Some approach (see [2] for example) assumes that the nodes are cooperating, so an overloaded node can pass some of his tasks to another one. When no such “helper” node is available, special centralized nodes are used as temporary load-balancers. Other solutions force the nodes to compete with each other. In the case of approach presented in [3], each node has a limited budget and tries to obtain tasks by buying them (similar to a real-life auction). In this way tasks can change nodes, but they must return to the initial node in the end. The nodes are concerned only with their own benefit, but this, in turn, results in better performance on the whole system.

In this paper we focus on the last of mentioned types of request dispatching, where all load-balancing and scheduling is done by single dedicated network component (or more generally, one such component for every back-end servers group), which simplifies the problem. In this case some some approaches exist as well. For example Cheng *et al.* in [4] tried to balance the load and meet QoS conditions (namely avarage bandwidth and latency) by using software adaptation mechanisms. Changes in the system's parameters are detected and the certain adaptation actions are carried out. These actions include: starting-up new servers, shutting down idle servers and switching users to another server group. This allows to increase the chosen parameters of QoS without the need to directly interfere with the process of dispatching tasks. Aside from that, the are some approaches that propose solutions beetwen centralized and distributed tasks dispatching. One such example was presented in [5], where the load-balancer is seemingly centralized, but the nodes inside the network of workstation or a computer cluster can exchange data or tasks. That work also analyses the differences beetwen centralized and distributed approach to load-balancing, as well as making distinction beetwen global and local load-balancing, where each local group of nodes is managed by a single centralised task dispatcher.

When optimization criteria are considered, most of above solutions assumes only one criterion such as: mean response time, makespan or power usage. Multi-criteria approaches are much less common, but they still appear. For example: Ben-Bassat and Borovits [6] consider two separate criteria: maximum jobs executed per unit of time and the minimum idle time per machine. They however consider the third criterion which is a combination of the previous ones. As another example, Garg and Singh [7] consider two conflicting criteria: execution time (makespan) and total cost of workflow execution. They consider a grid system and solve the problem using Non-dominated Sort Particle Swarm Optimization (NSPSO) approach with the choice of the final solution left to the user.

OUR APPROACH

We consider a computer system with *M* machines acting as back-end servers (*i.e.* nodes) and a single machine acting as a task dispatcher, which performs load-balancing and scheduling. The back-end servers cannot exchange data beetwen themsleves and every one of them is connected to the dispatcher via a computer network. Next we have a set of *N* tasks (requests) to be executed. We assume that the tasks are executed in batch mode (without interaction from the user). All of them arrive at the dispatcher and have to ultimately be assigned to one of the back-end servers. Each task *j* has: arrive time A_j (*i.e.* time at which the task arrive in the system and the earliest time it can be distpatched and start execution), deadline time D_j (*i.e.* the latest time the task should complete without penalty) and execution time E_j (*i.e.* the time needed to complete the task). The back-end servers are identical, meaning each task can be executed on any server, the execution time is not dependent on the machine in our case. We also assume that the tasks cannot be suspended (wth the exception described below), restarted or switched to another machine.

Arrive and deadline times are easily established, but the execution time is harder to determine. Not many tasks have predictable execution time, so we assume that task with unknown execution time are given some time to finish (depending on the priority for example). If they are not completed by that time, they are suspended and scheduled once again later on. We also assume that the time needed to transfer the task's data to the back-end server and the response back to dispatcher is included in the task's execution time.

C_j is the time when the task *j* completes (*i.e.* it is send back to the client as response). $P_j = max(0,C_j - D_j)$ is the penalty of the task *j*, while $R_j = C_j - A_j$ is its response time. We also assume $L_j=1$ when tasks *j* is late (*i.e.* $P_j > 0$) and $L_j=0$ otherwise. Then we consider minimizing the following criteria:

1. Mean penalty of all tasks: $\frac{1}{N}\sum_{i=1}^{N} P_j \rightarrow min$.
2. Mean response of all tasks: $\frac{1}{N}\sum_{i=1}^{N} R_j \rightarrow min$.
3. Makespan (*i.e.* maximal completion time of all tasks): $max\ C_j \rightarrow min$.
4. Number of late tasks: $\sum_{i=1}^{N} L_j \rightarrow min$.

5. Maximal penalty: $max\ P_j \rightarrow min$.
6. Maximal response time: $max\ R_j \rightarrow min$.

These criteria are high-level (compared to low-level network QoS parameters like bandwidth) and can be used to specify the requirements of end-user, as well as the system's managers. Some of above criteria are important for the client (mean respone and mean penalty for example), while other have a greater meaning for system managers (makespan, maximal and mean penalty). In our research we use objective function combining two or three criteria and we use different sets of criteria.

The crucial assumption of our work is that the executed task are time-consuming, *i.e.* mean execution time of a task is high enough, so the dispatcher has enough time to perform more sophisticated scheduling algorithms. This situation doesn't occur when tasks consist of simple requests like HTTP website serving, so we have to perform little calculations or serve single file up to 1 MB. However, we can also consider complex engineering calculations or simulations (especially now, when many services are migrating inside computing clouds and are thus performed remotely). In that case the tasks may take minutes or hours to complete and our approach becomes applicable. Therefore we add incoming tasks to a queue of non-dispatched tasks and wait for the moment when further dispatch delay is no longer acceptable. Then we use a memetic algorithm based on the Local Search Elitist Non-dominated Sorting Genetic Algorithm (or LSNSGA-II) to find a schedule for all accumulated tasks.

ALGORITHM DESCRIPTION

In paper [8] Deb *et al.* suggested an Elitist Non-dominated Sorting Genetic Algorithm. Based on the non-dominated sorting GA (NSGA), criticized for high computational complexity of non-dominated sorting, lack of elitism and need for specifying the sharing parameter, they modified the approach to alleviate those difficulties. By applying fast non-dominated sorting, density estimation and crowded comparison operator it allowed to lessen the computational complexity and guide the selection process of the algorithm towards a uniformly spread out Pareto-optimal front, *i.e.* Figure 1.

Fast non-dominated sorting divides solutions, obtained by current iteration of memetic algorithm, into Pareto-frontiers (see Figure 2 for example) with at most $O(mN^2)$ computations. Each solution is assigned two features: (a) the number of times the solution

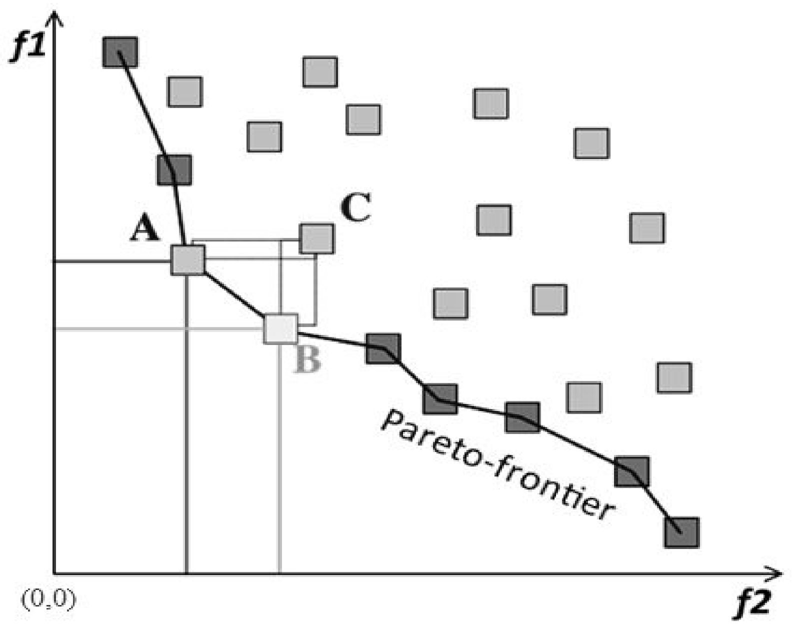

Fig. 1. – Non-dominated solutions – Pareto-frontier

was dominated (n_i) and (b) a collection of solutions dominated by the solution. To begin with, all of the solutions, which were not dominated, are moved from the solutions collection to the first Pareto-frontier. For each solution in the frontier, we check the set of solutions it dominates and for each member of that set we decrease the number n_i. Now all of the new non-dominated solutions, from the remaining collection, are moved to the second Pareto-frontier and again we decrease number of times their members were dominated.

In order to estimate the density of solutions surrounding a particular point in its front the average distance of the two points on either side of this point along each of the objectives is taken. This quantity serves as an estimate of the size of the largest rectangle enclosing that particular point without including any other point in the population. Deb *et al.* called this the crowding distance [8]. Figure 3. shows the crowding distance of the *i*-th solution in its front, marked with solid circles, as the average side-length of the rectangle, which is shown with a dashed box.

The crowded comparison operator guides the selection process at the various stages of the algorithm towards a uniformly spread out Pareto-optimal front [8]. That is, between two solutions with differing non-domination ranks it is preferred to take the point with the lower rank. Otherwise, if both the points belong to the same front then we prefer the point which is located in a region with lesser number of points, meaning that the size of the rectangle inclosing it is larger.

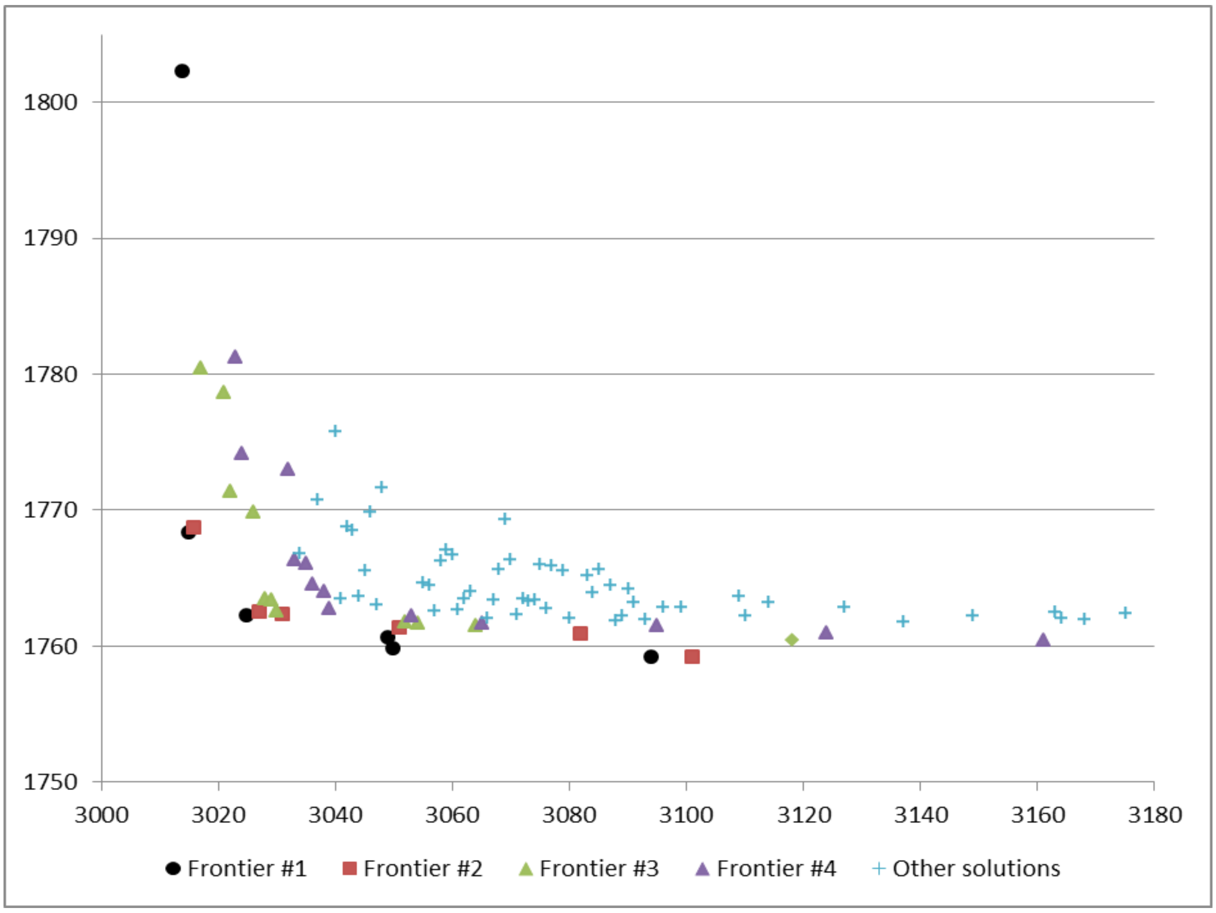

Fig. 2. – Sample of Pareto-frontiers from production scheduling

Developing our memetic algorithm we decided to use fast non-dominated sorting, in order to decrease computational complexity, and used sum of standardized criteria values as an order evaluator in each frontier, which allowed us to select one multi-criteria solution without involving a decision maker or analyzing previously made decisions. This way we eliminated additional computational time needed to provide a solution. Each solution represents a number of job sets, equal to the number of nodes, and is submitted to genetic operations, *i.e.* mutation, crossover and selection. We used a partially matched crossover (PMX) scheme and swap functions for mutation purposes [9], additionally we used tournament selection on randomized solutions list. Fitness value is based on frontier number, the lower the better, and standardized criteria values.

Diversified job sets, delivered by operators of genetic algorithm, are not scheduled and don't provide us with optimal solutions. Consequently, for each solution in offspring population the constructional algorithm is performed on every job set, one per node, in order to enhance the offspring. It uses a set of jobs assigned to a node and constructs a schedule from scratch minimizing (maximizing) criteria functions values. We based that method on the idea of job insertions [10]. We begin with inserting jobs with the lowest deadline or the highest job processing time value. Each job is inserted into all available positions in partial schedule and the one with lowest criteria value is selected [11], then another jobs are fitted into schedule until there are no unassigned jobs left.

In order to enhance our algorithm's performance, four constructional algorithm's solutions are inserted into initial population. This way, our algorithm can't perform worse than other know algorithms and is guided towards good solutions from the beginning. It means, we can perform less iterations of the algorithm and still work well. We prepared the algorithm to be flexible and the population size, as well as number of iterations, can be increased or decreased in order to accordingly provide better solutions with higher probability or speed the algorithm up if needed.

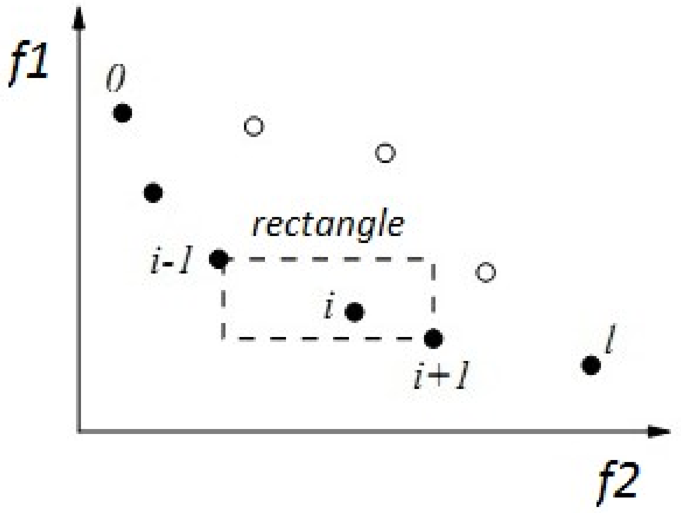

Fig. 3. – Crowding distance calculation

Our algorithm works on-line, but unless there are at least few jobs in the awaiting list, it doesn't start the memetic part and only distributes jobs between unoccupied nodes. Subsequently, it is an offline algorithm applied to specific online scheduling problem. When it's actuated it works with constrains of partially occupied nodes and takes their busy time into consideration while computing criteria values. During tests, for each problem instance, the algorithm had run multiple times for different subsets of calculated jobs, added to the buffer throughout the work time of nodes.

RESEARCH

We compared the results of our memetic algorithm (MA) with the solutions supplied by four constructive algorithms:

1. Round Robin (RR). Dispatches the task as soon at it arrives, assigning it to the next machine on the list (when there are no more machines, the machine counts goes back to the first machine).
2. Least Busy Machine (LBM). Dispatches the task as soon as it arrives, assigning it to the least busy machine. In our case it means machine that has the lowest estimated completion time of all machines.
3. Shortest Job First (SJF). Adds incoming tasks to tasks list when they arrive. Dispatches all accumulated tasks only when there are empty machines. It sorts the list of tasks by their execution time in ascending order, so the shortest tasks are dispatched first. Then the SJF algorithm assigns each task along the list to the least busy machine at the time.
4. Earliest Deadline First (EDF). Works exactly like the SJF algorithm, but sorts the tast according to their deadline time, so the most important tasks are disaptched first.

In order to compare the quality of the solutions supplied by the algorithms, we consider following methodology. Let us consider the problem instance *I* (a set of tasks). First for each algorithm A_i we calculate two numbers: $B(A_i)$ and $W(A_i)$ as follows:

$$B(A_i)=\sum_{j=1}^{J}\sum_{c=1}^{C} d(A_i, A_j, c) \quad (1)$$

$$W(A_i)=\sum_{j=1}^{J}\sum_{c=1}^{C} d(A_j, A_i, c) \quad (2)$$

where:

- *C* is the number of criteria considered,
- J is the number of algorithms (since we compare 5 algorithms, it is always equal to 5),
- $d(A_i, A_j, c)=1$ when algorithm i is better than algorithm j over the criterion c (*i.e.* $A_i(c)<A_j(c)$) and $d(A_i, A_j, c)=0$ otherwise.

In short: $B(A_i)$ denotes how many times a given algorithm was better than other algorithms (according to the given criterion) and $W(A_i)$ denotes how many times the same algorithm was worse than other algorithms. Now we calculate $Q(A_i)=B(A_i)-W(A_i)$. Thus for a given instance *I* we end with single quality parameter $Q(A_i)$ for each tested algorithm.

We construct our instances as follows: arrive time A_j of task *j* is generated randomly using the normal distribution approximation from the Box-Muller transform. Moreover $A_j \in [0, f \cdot N]$, where *N* is the number of tasks and coefficient *f* is in range $[0.8, 1.2]$. Execution time E_j is generated randomly using the uniform distribution. E_j is in range $[10,50]$. Deadline time $D_j = A_j + k \cdot E_j$, where *k* is generated randomly from uniform distribution in range $[1.1, 1.4]$. We considered instances with 20, 50, 100 and 200 tasks (*i.e.* the number of tasks that are accumulated and dispatched at once). The number of machines ranged from 5 to 15. For each such instance type we have prepared 10 instances. As for criteria we considered four different criteria sets:

1. Mean penalty $\bar{P}$ and mean response $\bar{R}$.
2. Mean response $\bar{R}$ and maximal completion time $max\ C_j$.
3. Maximal penalty $max\ P_j$ and mean response $\bar{R}$.
4. Maximal response $max\ R_j$, number of late tasks $|L|$ and maximal completion time $max\ C_j$.

For example the results of $Q(A_i)$ paremeter research in the case of 10 instances with 50 tasks and 5 machines with the last criteria set (maximal response, number of late tasks and maximal completion time) are shown in Table 1.

Table 1. – $Q(A_i)$ results for instances with 50 tasks and 5 machines

Algorithm	**RR**	**LBM**	**SJF**	**EDF**	**MA**
Instance 1	0	-6	0	3	3
Instance 2	1	-9	2	3	3
Instance 3	-4	4	-4	2	2
Instance 4	-11	5	3	0	3
Instance 5	-11	-3	5	4	5

Instance 6	-11	-1	6	0	6
Instance 7	-2	3	0	-4	3
Instance 8	-11	-3	5	4	5
Instance 9	-8	3	4	-3	4
Instance 10	-2	3	0	-4	3
Total	**-59**	**-4**	**21**	**5**	**37**

The results show that constructive algorithms are unreliable when multiple criteria are specified, as the number of times an algorithm dominates other algorithms or is dominated by them changes abrutply with instance even though all instances are of the same category. There are however criteria sets (like mean response and mean penalty set) when one constructive algorithm is visibly dominating the other constructive ones. Above table also shows the summary of each algorithm and that our memetic algorithm is the most successful in generating non-dominated solutions.

Next we present the research of $Q(A_i)$ parameter in relation to the instance category and criteria set. Results are shown in Tables 2 and 3. First we consider different criteria sets (Table 2). In all cases, the Round Robin algorithm appears to be a poor choice and is easily dominated by other algorithms. Looking at the first criteria set (*i.e.* mean penalty and mean response), it seems that Shortest Job First is performing well and outclasses both Earliest Deadline First and Least Busy Machine algorithms. However each subsequent criteria set proves, that EDF and LBM are dominated less and less often and, finally, when 3 criteria are specified, they even manage to generate solutions of quality equal to SJF algorithm. That is because constructive algorithms lack the explicit definition of objective function, and thus lose their reliability when multiple criteria are specified. Our memetic algorithm,

Table 2. – Summarized $Q(A_i)$ results for all instances in relation to criteria set.

Algorithm	**RR**	**LBM**	**SJF**	**EDF**	**MA**
Set 1	-366	-170	270	-32	298
Set 2	-355	-59	229	-39	224
Set 3	-276	-12	64	78	146
Set 4	-307	59	61	60	127

however, has no such weakness and easily outclasses all constructive algorithms. The only exception is the SJF algorithm, which is dominated less often. In the case of the second criteria set (*i.e.* mean response and max completion) the SJF algorithm seems to dominate over MA. This rare situation can occur when MA improves one criterion (so it is better than other 4 algorithms on this criterion), but in the process worsens the other one so much, that it becomes worse than all 4 algorithms. In result the $Q(MA)=0$. On the other hand SJF is better than 3 algorithms on the first criterion, while it loses only to MA on the second criterion, thus $Q(SJF)=2$. In result SJF has better $Q(A_i)$, even though our algorithm has better total objective function value.

Table 3. – Summarized $Q(A_i)$ results for all instances in relation to instance category.

Algorithm	**RR**	**LBM**	**SJF**	**EDF**	**MA**
20 tasks, 5 machines	-216	-69	134	-8	159
50 tasks, 5 machines	-277	-80	141	35	181
100 tasks, 5 machines	-270	-8	116	22	140
100 tasks, 10 machines	-266	-32	151	-21	168
200 tasks, 15 machines	-275	7	82	39	147
Total	**-1304**	**-182**	**581**	**67**	**795**

In Table 3 we present the values of $Q(A_i)$ parameter in relation to different instance categories. This time all criteria sets are summarized, showing that MA is the most successful at dominating other algorithms, surpassing SJF by at least 10% and over 30% on avarage. We alse notice, that when our 4 criteria sets are considered, the algorithms can be ordered by their total $Q(A_i)$ value with MA being the best (but most complex) algorithm and RR being the worst, but simplest solution. Moreover, the order of algorithms remains constant in relation to the instance category.

CONCLUSIONS

In this paper we presented our approach to multi-criteria network scheduling of time-consuming tasks with the criteria specified by the end-user (as part of Quality of Service) or by the system's managers. We used memetic algorithm based on the Local Search Elitist

Non-dominated Sorting Genetic Algorithm and compared it to four known constructive algorithms in terms of their ability to generate non-dominated solutions. The results prove that our algorithm is significantly better that the SJF algorithm and easily outclasses other algorithms. The main advantages of our approach are: (a) explicit objective function, (b) use of constructive algorithms for the creation of the initial population, and (c) great flexibility. Our MA algorithm can be easily modified to use different crossover and mutation operators, as well as different methods of selection. Adding support for weighted criteria is also possible. The size of the population or the number of iterations can be changed depending on the amount of time left for scheduling. Even when the time runs out, the algorithm can be stopped and the results of the last iteration can be used to obtain solutions. Finally, the solutions obtained are always at least as good as the solutions provided by the constructive algorithms and they can be often improved. However, the level of improvement depends heavily on the chosen criteria and the tasks set's parameters (*i.e.* the profile of the tasks), so the efficiency of our approach will change with different systems and needs of the users.

REFERENCES

[1] V. Cardellini, M. Colajanni, P. S. Yu. *Dynamic Load Balancing on Web-Server Systems. IEEE Internet Computing*, 3(3):28–39, May 1999.

[2] G. D. Jain P. *An Algorithm for Dynamic Load Balancing in Distributed Systems with Multiple Supporting Nodes by Exploiting the Interrupt Service. International Journal of Recent Trends in Engineering*, 1(1):232–236, May 2009.

[3] D. Ferguson, Y. Yemini, C. Nikolaou. *Microeconomic algorithms for load balancing in distributed computer systems. 8th International Conference on Distributed Computing Systems*, pp. 491–499, 1988.

[4] S.-W. Cheng, D. Garlan, B. R. Schmerl, J. a. P. Sousa, B. Spitznagel, P. Steenkiste, N. Hu. *Software Architecture-Based Adaptation for Pervasive Systems. Proceedings of the International Conference on Architecture of Computing Systems: Trends in Network and Pervasive Computing*, ARCS'02, pp. 67–82, London, UK, UK, 2002. Springer-Verlag.

[5] M. J. Zaki, W. Li, S. Parthasarathy. *Customized Dynamic Load Balancing for a Network of Workstations. Journal of Parallel and Distributed Computing*, 43:156–162, 1995.

[6] M. Ben-Bassat, I. Borovits. *Computer network scheduling*. *Omega*, 3(1):119–123, February 1975.

[7] R. Garg, A. K. Singh. *Multi-objective workflow grid scheduling based on discrete particle swarm optimization*. *Proceedings of the Second international conference on Swarm, Evolutionary, and Memetic Computing - Volume Part I*, SEMCCO'11, pp. 183–190, Berlin, Heidelberg, 2011. Springer-Verlag.

[8] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan. *A fast and elitist multiobjective genetic algorithm: NSGA-II*. *Evolutionary Computation, IEEE Transactions on*, 6(2):182 – 197, apr 2002.

[9] E. Zitzler, L. Thiele. *Multiobjective Optimization Using Evolutionary Algorithms - A Comparative Case Study*. *Proceedings of the 5th International Conference on Parallel Problem Solving from Nature*, PPSN V, pp. 292–304, London, UK, UK, 1998. Springer-Verlag.

[10] E. Nowicki, C. Smutnicki. *A fast taboo search algorithm for the job shop problem*. *Manage. Sci.*, 42(6):797–813, June 1996.

[11] M. Pinedo. *Scheduling: Theory, Algorithms and Systems*. Prentice Hall, second edition.