

Artificial Life Techniques for Load Balancing in Computational Grids

Riky Subrata

Albert Y. Zomaya

Bjorn Landfeldt

Advanced Networks Research Group
School of Information Technologies,
University of Sydney, NSW 2006 Australia

Riky Subrata

E-mail address: efax@it.usyd.edu.au

Albert Y. Zomaya

Tel: +61 2 9351 3424

Fax: +61 2 9351 3838

E-mail address: zomaya@it.usyd.edu.au

Bjorn Landfeldt

E-mail address: bjornl@it.usyd.edu.au

ABSTRACT

Load balancing is a very important and complex problem in computational grids. A computational grid differs from traditional high performance computing systems in the heterogeneity of the computing nodes and communication links, as well as background workloads that may be present in the computing nodes. There is a need to develop algorithms that could capture this complexity yet can be easily implemented and used to solve a wide range of load balancing scenarios. Artificial life techniques have been used to solve a wide range of complex problems in recent times. The power of these techniques stems from their capability in searching large search spaces, which arise in many combinatorial optimization problems, very efficiently. This paper studies several well-known artificial life techniques to gauge their suitability for solving grid load balancing problems. Due to their popularity and robustness, a Genetic algorithm (GA) and tabu search (TS) are used to solve the grid load balancing problem. The effectiveness of each algorithm is shown for a number of test problems, especially when prediction information is not fully accurate. Performance comparisons with Min-min, Max-min, and Sufferage are also discussed.

1 INTRODUCTION

The computational grid is a promising platform that provides large resources for distributed algorithmic processing [8]. Such platforms are much more cost-effective than traditional high performance computing systems. However, computational grid has different constraints and requirements to those of traditional high performance computing systems. To fully exploit such grid systems, resource management and scheduling are key grid services, where issues of task allocation and load balancing represent a common problem for most grid systems.

The load balancing mechanism aims to equally spread the load on each computing node, maximizing their utilization and minimizing the total task execution time [33]. In order to achieve these goals, the load balancing mechanism should be ‘fair’ in distributing the load across the computing nodes. This implies that the difference between the heaviest-loaded node and the lightest-loaded node should be minimized. In this paper, we will use a standard performance metric – the application *makespan*, to evaluate our approaches and other approaches that have been proposed in the literature. The application makespan is defined in this study as the amount of time taken from when the first input file is sent for computation (to a computing node), to when the last task is completed by a computing node.

In general load balancing algorithms can be classified as *centralized* or *decentralized*, and *static* or *dynamic*. In the centralized approach (eg. [28]), one node in the system acts as a scheduler and makes all the load balancing decisions. Information is sent from the other nodes to this node. In the decentralized approach (eg. [35]), all nodes in the system are involved in the load balancing decisions. It is therefore very costly for each node to obtain and maintain the dynamic state information of the whole system. Most decentralized approaches let each node obtains and maintains only partial information locally to make sub-optimal decisions.

Static load balancing algorithms (e.g. [25]) assume all information governing load balancing decisions that can include the characteristics of the jobs, the computing nodes and the communication network) are known in advance. Load balancing decisions are made deterministically or probabilistically at compile time and remain constant during runtime. The static has one major disadvantage – it assumes that the characteristics of the computing resources and communication network are all known in advance and remain constant. Such an assumption may not apply to a grid environment. In contrast, dynamic load balancing

algorithms attempt to use the runtime state information to make more informative load balancing decisions. Undoubtedly, the static approach is easier to implement and has minimal runtime overhead. However, dynamic approaches may result in better performance.

One simple load balancing algorithm is the *Best-fit* algorithm. In this algorithm, tasks are assigned according to their order in the queue. Each task is then scheduled to available computing nodes based on the completion time offered by the nodes; the node that completes the task the fastest (taking into account its current load) is chosen. In this study, tasks are assigned/scheduled to computing nodes as soon as the computing nodes are available. Once a task is assigned to a node, it will be executed on that node and will not be re-assigned to another node. If there are tasks in the queue and no available computing nodes, the scheduler will wait until there is an available node, at which time the scheduling of tasks in the queue will continue.

Three load balancing algorithms, which can be implemented statically or dynamically in a grid environment, were proposed in [22, 29]. These are *Min-min*, *Max-min*, and *Sufferage*. Each of the three algorithms assigns tasks iteratively to the computing nodes. Beginning from a set of tasks to schedule, a task is chosen, assigned to a node, and removed from the set. This process continues until all the tasks are assigned or ‘enough’ tasks have been assigned to the nodes. The set of tasks that are included for scheduling include unassigned (have not been assigned a computing node) tasks, as well as assigned unexecuted tasks. A task is chosen from the set in the following way. First, the completion time for each task in each processor is computed. A *minimum completion time* (MCT) for each task is then computed. Finally, a *metric* is computed for each task, and the task with the ‘best’ metric is assigned to the processor corresponding to the task’s MCT. The three algorithms differ in the way the tasks’ metric are computed.

In the Min-min algorithm, a task with the minimum MCT is chosen. This means tasks with the lowest completion times are given priority. In Max-min, a task with the maximum MCT is chosen. Finally, in Sufferage, a sufferage value is calculated and used as the metric. In this study, the sufferage value for a task is defined as the difference between its best MCT and its second best MCT. The task with the highest sufferage value is then chosen.

In this paper, we propose the use of genetic algorithm (GA) and tabu search (TS) to find efficient solutions to the grid load balancing problem. A Sufferage algorithm is combined with GA, as well as TS, which improves the overall performance of the algorithms. A comparison on the performance improvement

is also investigated.

The next section is an overview of the system model including the grid and communication model that we are using. This is followed by the development of a genetic algorithm and tabu search algorithms to solve the grid load balancing problems. The results section provides a number of detailed simulations that shows the applicability of the proposed approaches.

2 SYSTEM MODEL

We assume that the computational grid system consists of a set of sites S connected by a communication network. In general, each site may contain multiple number of computing nodes, and each computing node may have single or multiple processors. The processors in the nodes are heterogeneous, meaning they may have different processing power. Without loss of generality and to emphasize our main ideas, we assume each site has one computing node equipped with a single processor; the processors in the different computing nodes have different processing power.

A computing node in the grid may also have a local user (or multiple local users) that uses the node for other computations (that is, the node is not a dedicated node). As such, at any one time a computing node may have background workload associated with it, which will affect the completion time of the grid jobs assigned to it. Each computing node has a task-time queue that can hold a certain number of tasks/jobs. Tasks are queued as long as the queue is not full. To this end, each computing node sends a reply message, after it has received a task or completed a task, indicating if it can receive more tasks. The scheduler sends one task at a time to the computing nodes that are available to receive tasks, as long as there are tasks in the scheduler's queue that have been assigned to a computing node.

The sites s_1, \dots, s_n in S is fully interconnected, meaning that there exists a communication path between any two sites (s_i, s_j) in S . Inter-site communication is done via message passing, and the underlying network protocol guarantees that messages are received by the intended recipient in the order they are sent. Our communication model represents network performance between a site s_i to a site s_j using two parameters – a transmission delay TD_{ij} representing start-up cost and contention delays at intermediate links on the path from s_i to s_j , and a data transmission rate BW_{ij} representing the bandwidth available on the path from s_i to s_j . For a message of size m to be transmitted from site s_i to s_j , the transmission time is then given by $TD_{ij} + m/BW_{ij}$. TD_{ij} and BW_{ij} can be calculated from analytical models or historical information, or

dynamically forecasted by facilities such as the Network Weather Service (NWS) [39].

2.1 Application model

One common application model for the computational grid is the bag-of-tasks application. In this application model, an application consists of a set of independent tasks with no required order of execution. The tasks are of different computation sizes and communication sizes, meaning each task requires a different computation time and data transmission time for completion. Bag-of-tasks applications can be classified into two categories: data intensive and computationally intensive. In this paper, computationally intensive bag-of-tasks (CBoT) application are of particular interest. A CBoT application is characterized by a high ratio of computation time over communication time.

Some researchers have considered job migration (migration of partly executed jobs) in their load balancing algorithms. However, job migration is far from trivial in practice. It involves collecting all system states (e.g. virtual memory image, process control blocks, unread I/O buffer, data pointers, timers etc.) of the job, which is large and complex. Several studies (e.g. [27, 42]) have shown that: (1) job migration is often difficult to achieve in practice, (2) the operation is generally expensive in most systems, and (3) there are no significant benefits of such a mechanism over those offered by non-migratory counterparts. As such, we do not consider migration of partly executed jobs in this paper. Instead, a job that is currently executing on a computing node may be cancelled, and restarted on another node.

3 GENETIC ALGORITHM

A genetic algorithm (GA) [16, 30, 31, 38] is a biologically inspired optimization and search technique developed by Holland [21]. Its behavior mimics the evolution of simple, single celled organisms. It is particularly useful in situations where the solution space to be searched is huge, making sequential search computationally expensive and time consuming.

GA is a type of guided random search technique, able to find ‘efficient’ solutions in a variety of cases. Efficient here is defined as solutions that, though they may not be the absolute optimal, is within the constraints of the problem, and is found in a reasonable amount of time and resources. To put another way, the effectiveness or quality of a GA (for a particular problem) can be judged by its performance against other known techniques – in terms of solutions found, and time and resources used to find the solutions. That is,

much like everything else in life, we judge by relative performance or benchmark – have we performed better than our competitors? It should also be pointed out that in many cases (much like everything else in life) the absolute optimal is not known. In this respect, GA has shown itself to be extremely effective in problems ranging from optimizations to machine learning [23, 30, 31, 41].

Some common terminology used in GA is described below. The relationships between the different ‘units’ used in the GA system are shown in Figure 1.

Gene A basic entity in GA, a single gene represents a particular feature or characteristic of *individuals* in the GA system.

Allele A value associated with a gene. For example, biologically speaking, a gene for eye color may have an allele value of ‘brown’.

Chromosome An *individual* in the GA system, consisting of a *string* (or collection) of genes. Each chromosome represents a *solution* (usually an encoding of the solution), or a parameter set, for the particular problem.

Locus The *position* of a gene in the chromosome.

Population Collection of chromosomes.

The gene in GA takes on a value from an alphabet of size $r \in \mathbb{Z}^+$. Each chromosome / string then consists of a series of n genes:

$$\tilde{s} = \{b_1, b_2, \dots, b_n\} \quad (1)$$

resulting in a solution space of size r^n . The most common, and generally most effective [16] string representation is the binary alphabet $\{0, 1\}$. In this case, each gene takes on a possible value of 0 or 1:

$$b_i \in \{0, 1\}, \quad i = 1, 2, \dots, n \quad (2)$$

whereby the solution space is now of size 2^n . In most cases, the use of binary gene (resulting in a binary string) results in longer chromosome length (to represent the parameter set, or solutions, to the problem). At the same time, this results in more genes available to be ‘exploited’ by the GA, resulting in better performances in many cases [16, 31].

The evolution of the GA population from one generation to the next is usually achieved through the use of three ‘operators’ that are fundamental in GA: *selection*, *crossover*, and *mutation*.

3.1 Selection

Selection is the process of selecting chromosomes (in the population) from the current generation, for ‘processing’ to the next generation. For this purpose, highly fit chromosomes / individuals are usually given a higher chance of being selected more often, thereby producing more ‘offspring’ for the next generation.

Some common techniques to achieve this objective include:

- Fitness-proportionate selection.
- Rank selection.
- Tournament selection.
- Elitism; this method is used in conjunction with the other selection methods.

In the fitness-proportionate selection method, the number of times an individual is selected for reproduction is proportional to its fitness value, as compared to the average fitness of the population. One simple method is to set the *expected* ‘count’ of each individual to the following:

$$E(i) = \frac{f_i}{\bar{f}} \quad (3)$$

where $E(i)$ denotes the expected count of an individual i , f_i denotes the fitness value associated with individual i , and \bar{f} denotes the average fitness of the population. One disadvantage of using the fitness-proportionate selection is that too much emphasis may be put on individual’s fitness as the selection criteria. Such emphasis can cause problems in some cases. As the initial population of the GA is usually created randomly, typically initially (and early on in the search) the fitness variance in the population is high. As such, if initially the population contains a few highly fit (but sub-optimal) individuals, and the rest of the population are relatively ‘low fit’ in comparison, then these highly fit individuals will dominate the selection process, due to their high selection probabilities. These highly fit individuals (and their descendants) will multiply quickly, and in a few generations will have dominated the population – resulting in a quick and probably premature convergence. In effect, exploration of the search space by the genetic algorithm (GA) is restricted, due to the fitness-proportionate selection process. On the other hand, later on in the iteration when the population has a low variance (the individuals in the population have ‘similar’ fitness values), then the

selection process will result in all individuals having around the same selection probabilities. Since ‘good’ and ‘bad’ individuals have around the same chance of being selected, this results in a ‘stale’ population, whereby the average fitness of the population is not improved in further generations.

To overcome the problems of fitness-proportionate selection, whereby the ‘evolution’ of the GA depends highly on the population’s fitness variance, other selection methods have been proposed. In the rank-based selection scheme, individuals in the population are first sorted according to their fitness values. Each individual’s rank in the population – instead of its fitness value, is then used to ‘influence’ the selection process. One simple, non-linear rank-based selection scheme is the following [40]:

$$p_i = \frac{R(i)}{\sum_{j=1}^n R(j)} \quad (4)$$

where p_i denotes the selection probability of individual i , n denotes the population size, and $R(i) \in [1, n]$ denotes the rank of individual i . In this case, $R(n)$ represents the fittest (best) individual, and $R(1)$ the worst individual in the population.

Finally, the tournament selection scheme uses the ‘idea’ of the rank-based selection scheme described above, but in a computationally more efficient way. Tournament selection doesn’t require the whole population to be sorted according to their fitness values. Further, the tournament selection scheme can be implemented in parallel. In the non-probabilistic tournament selection, a certain number k of individuals is chosen at random from the population. The fittest individual in this ‘group’ is then selected (given the right to reproduce). The individuals in the group are then ‘returned’ to the population, and can be selected again. This process is repeated n (where n is the population size) times. Typically, k is set to two. In this case, the fitter of the two individual is considered the ‘winner’, and is selected.

While in the above selection scheme the fitter of the individuals always win, this is not always the case in the probabilistic tournament selection scheme. In the basic probabilistic tournament selection, two individuals are chosen at random from the population. The two individuals then ‘compete’ in a virtual tournament, whereby the fitter individual has a $p_s \in [0, 1]$ chance of winning (e.g. $p_s = 0.8$). The winner is then selected (given the right to reproduce). As before, this process is repeated until the required number of individuals has been selected for reproduction.

Elitism is usually used in conjunction with the other selection methods. In its simplest form, the best σ individuals are ‘cloned’ and guaranteed to be present, in its current form, in the next generation. Such method is used to ensure that the best individuals are not lost, either because they are not selected in the selection procedure, or because their genes have been changed by crossover and mutation. Without elitism, the best solution (represented by one or more individuals in the population) available may vary wildly from generation to generation; with elitism, however, the best solution available in the current generation is guaranteed to be better or at least the same, as the previous generation. The use of elitism improves the performance of the genetic algorithm in many cases [7, 17, 18, 24, 26, 31, 32, 34].

3.2 Crossover

Once individuals / chromosomes have been selected (for reproduction), crossover is applied to the chosen individuals. The crossover operator usually operates on two individuals / parents, to produce two children. The crossover operator is one of the distinguishing features of a genetic algorithm, and its use ensures that characteristics of each parent are inherited in the children.

Common crossover methods that are readily used for binary gene include *one-point* crossover, *two-point* crossover, and *uniform* crossover. Which of these crossover methods is the most appropriate to use depends on the particular problems, in particular the chromosome encoding (to represent the problem set), and fitness function used. Note though the two-point crossover and uniform crossover is commonly used in recent genetic algorithm applications [31].

In uniform crossover gene exchanges can occur at any position on the chromosome. For example, if we have two parents 000000 and 111111, then to produce two offspring we go through each of the six genes of one of the parent. For each gene, there is a probability p_s (e.g. $p_s = 0.7$) that this gene will be exchanged with the other parent. As such, if an exchange happens at position 2 and 5, then the two offspring produced will be 010010 and 101101.

3.3 Mutation

While the crossover operator works on a pair (or more) of chromosomes / individuals to produce two (or more) offspring, the mutation operator works on each individual offspring. The mutation operator helps prevent early convergence of the genetic algorithm (GA) by changing characteristics (allele) of

chromosomes in the population. Such changes in the chromosomes also results in the GA's ability to 'jump' to far away solutions, hopefully to unexplored areas of the solution space. Clearly though, the *mutation rate* of the chromosomes should not be set too high, as too much mutations has the effect of 'changing' the guided random search of the GA to a purely random search.

3.4 Genetic algorithm framework

The GA used for efficient solutions to the load balancing problem, considering the points discussed above, is shown below.

ALGORITHM 1 (GA FOR LOAD BALANCING) :

Input: Parameters for the GA.

Output: Population of solutions, \mathbf{P} . Each of these solutions can be used as a task schedule.

Begin

Initialize the population, \mathbf{P} .

Evaluate \mathbf{P} .

While stopping conditions not true **do**

Select *Elite* in \mathbf{P} consisting of k ($1 \leq k \leq \text{population size}$) best individuals.

Apply *Selection* from individuals in \mathbf{P} to create $\mathbf{P}_{\text{mating}}$, consisting of ($\text{population size} - k$) individuals.

Crossover $\mathbf{P}_{\text{mating}}$.

Mutate $\mathbf{P}_{\text{mating}}$.

Copy the whole individuals of $\mathbf{P}_{\text{mating}}$ to \mathbf{P} , replacing the worst ($\text{population size} - k$) individuals in \mathbf{P} .

Evaluate \mathbf{P} .

If escape condition true **then** *Escape*.

End.

The parameters for the GA are shown in Table 1. First, a fixed number of tasks are considered for scheduling. This is to stop the scheduler from scheduling an excessive number of tasks at once; this also limits the computation time required by the scheduler. In this study, a window size of eight times the number of nodes available is used. The window represents all the tasks in the queue that will be considered for scheduling; If the number of tasks in the queue is less than the window size then *TaskSet* consists of all the tasks in the queue; otherwise *TaskSet* consists of all the tasks in the window. Note that a task remain in the scheduler's queue until it is completed; tasks that have been sent or are executing on computing nodes still

remain in the queue, as they may be re-allocated to another node.

A population of solution is then created. In our case, a chromosome population of twice the size of *TaskSet* is used. Each chromosome represents a valid task schedule. A binary string representation is used for the chromosomes. For example, if we have four tasks to schedule and three nodes, one possible string is: 0101|1000|0010, which says that task 2 and 4 is scheduled on node 1, task 1 is scheduled on node 2, and task 3 is scheduled on node 3. Note that each binary gene can have a value of either '0' or '1'. Note that the execution order is not coded into the solution, as this would result in a sharp increase in the search space the GA has to explore. In the experiments we have also found that, because load balancing/scheduling are done periodically, that is to say that the task schedules are continually refined, the execution order of tasks at any period does not greatly affect the overall makespan.

At initialization, the configuration for each chromosome is generated. The generation process is as follows. First, a Sufferage algorithm is used to generate an initial configuration for the chromosomes. A certain number (in this study we use 50%) of chromosomes are then mutated with mutation probability of 0.3. The mutation operator used is a custom bit mutation, whereby a task is randomly moved to another node. This initial population is then evaluated. After the initial evaluation, the GA goes into the 'evolution loop'. The GA continues to evolve until the stopping conditions are met. In this case, a fixed number of iterations/generations of 200 is used. The use of a fixed number leads to a greater predictability of the running time of the algorithm, as well as limiting the running time of the algorithm.

At each generation, the chromosomes/rules in the population are sorted in descending order, according to their fitness. In this experiment, the best σ rules are kept, and survive to the next generation intact. To this end, *tournament selection* is applied to the m chromosomes in the current population, with probability $p_s=0.8$, to create a mating pool of $m-\sigma$. In the tournament selection, two individuals are chosen at random from the current population P . The two individuals then 'compete' in a virtual tournament, whereby the fitter individual has a p_s chance of winning. The winner is then given the right to 'mate', proceeding in this case to the mating pool. This process is repeated until a specified number of individuals have been selected. It should be noted that the same individual can be selected more than once, leading to the individual 'mating' more than once.

To the chromosomes in the mating pool, a custom *uniform crossover* with probability $p_c=0.8$ is applied. In the crossover, only task assignment that is different in the two strings needs to be considered; For example, if

we have four tasks to schedule and three processors, two possible strings are 0101|1000|0010 and 1100|0010|0001. If a crossover occurs in the first and last position of the two strings then we have 1100|0000|0011 and 0101|1010|0000. After crossover, a custom *gene mutation* operator of probability $p_m=0.0005$ is applied. These m - σ chromosomes are then copied to the current population \mathbf{P} , which, together with the best σ chromosomes, forms the next generation of chromosomes.

Finally, the ‘escape’ condition is checked. Here, the escape condition is the number of generation since the last improvement in solution. If this number exceeds the maximum stale period $T_{stale}=20$, an escape operator is applied. In the escape operator, the population undergoes a massive gene mutation of $p_{mm}=0.2$.

4 TABU SEARCH

Tabu search (TS) [10, 11, 15] was first proposed in its current form by Glover [9]. It has been successfully applied to a wide range of theoretical and practical problems, including graph coloring, vehicle routing, job shop scheduling, course scheduling, and maximum independent set problem [1-6, 19, 20, 36, 37].

One main ingredient of tabu search (TS) is the use of *adaptive memory* to guide problem solving. One may argue that *memory* is a necessary component for ‘intelligence’, and intelligent problem solving. Tabu search uses a set of strategies and learned information to ‘mimic’ human insights for problem solving, creating essentially an ‘artificial intelligence’ unto itself – though problem specific it may be.

Tabu search also has links to evolutionary algorithms through its ‘intimate relation’ to scatter search and path re-linking [12, 14]. Numerous computational studies over the years have shown that tabu search can compete, and in many cases surpass the best-known techniques [13, 20]. In some cases, the improvements can be quite dramatic.

In its most basic sense, a tabu search can be thought of as a local search procedure, whereby it ‘moves’ from one solution to a ‘neighboring’ solution. In choosing the next solution to move to, however, tabu search uses memory and extra knowledge endowed about the problem. A basic tabu search algorithm is shown below.

ALGORITHM 2 (BASIC TABU SEARCH) :

Input: Parameters for the TS.

Output: A feasible solution to the problem.

Begin

 Generate an initial solution s .

While stopping conditions not true **do**

 Select next solution neighboring s .

 Update memory.

End.

For a given problem to be solved using tabu search, three things need to be defined: a set V of feasible solutions (for the problem), a *neighborhood structure* $N(s)$ for a given solution $s \in V$, and a *tabu list*, TL .

As shown in Algorithm 2, an initial solution s is chosen from the set of feasible solution V . This initial solution is usually chosen randomly. Once an initial solution is chosen, the algorithm goes into a loop that terminates when one or more of the stopping conditions are met. In the next step of Algorithm 2, the next solution s^{i+1} is selected from the neighbors of the current solution s^i , $s^{i+1} \in N(s^i)$. In a basic tabu search, all possible solutions $s \in N(s^i)$ is considered, and the best solution is chosen as the next solution s^{i+1} . Note, however, the use of such selection rule may result in the algorithm going in circles. At the very worst, the algorithm will go back and forth between two solutions. To avoid this, memory is incorporated into tabu search.

4.1 Tabu list

The simplest way such a memory can be used is to remember the last k solutions that have been visited, and to avoid these solutions. That is, the algorithm keeps a *list* of tabu solutions. The length of this *tabu list* may be varied, and a longer list will prevent cycles of greater length – a list of length k will prevent cycles of length $\leq k$. However, it may be impractical (e.g. time consuming to compare the solutions) to incorporate such memory. As such, *simpler* type of memories is usually incorporated, one of which is remembering the last k moves made. A move for a solution s^i can viewed as ‘changes’ applied to the solution s^i , to get to a new solution $s^{i+1} \in N(s^i)$. Generally, moves are defined so that they are *reversible*: for a move m from s^i to s^{i+1} , there is a *reverse* move m^{-1} from s^{i+1} to s^i . The tabu search can then keep a tabu list of the last k reverse moves made, and avoid making these moves.

Clearly though, such a list is not a perfect replacement for a solution list. While a move list is much easier

to use than a solution list, in many cases information is lost in that one cannot know for certain if a solution has in fact been visited in the last k moves. That is, the use of a move list does not guarantee that no cycle of length $\leq k$ will occur. In other cases, its use results in a restrictive search pattern – an unvisited solution may be ignored because a move to that solution is in the tabu list. Still, in other cases, the use of a move list results in both a loss of information, and a more restrictive search pattern. To partially overcome the restriction imposed by using a move list (for the tabu list), a tabu search usually incorporates *aspiration conditions*, whereby a tabu move (a move that is in the tabu list) is selected if it satisfies one or more of the aspiration conditions. One simple aspiration condition that is commonly used in tabu search is the following. If a tabu move leads to a solution that is better than the best solution found so far, then it should be selected. For further information on solution list and move list, refer to [10, 11, 15].

Finally, one other important feature of tabu search is that of *search intensification* and *search diversification*. For this purpose, a more elaborate memory structure that takes into account the *recency*, *frequency*, and *quality* of solutions and moves made so far is used.

4.2 Search intensification

In search intensification, exploration is concentrated on the vicinity, or ‘neighbors’ of solutions that have historically been found to be good. That is, search is concentrated around the *elite* solutions. Such search may be initiated when there is evidence that a region (on the solution space) may contain ‘good’ solutions.

During search intensification, a *modified* fitness function is used to ‘encourage’ moves or solution’s features that have historically been found to be good. For example, a ‘reward’ value may be added to the fitness function, so that certain moves or solution’s features become more attractive. Further, a modified neighborhood structure may also be used, in order to effectively combine good solution’s features.

Search intensification is usually carried out over a few numbers of iterations. If the process is not able to find a better solution than the best one found so far, a search diversification is usually carried out.

4.3 Search diversification

The search diversification process, unlike search intensification, attempts to spread out the search process to unvisited regions, and encourages moves or solution’s features that are *significantly* different from those that have been found. For this purpose, a modified fitness function is used, whereby *penalties* are applied to

certain moves or solution's features. For example, penalties may be given to frequently made moves, or common solution's features.

4.4 Tabu search framework

For the tabu search load balancing implementation, we need to define a neighborhood structure for a given solution S (where S represents a suitable task schedule, as described earlier). Here, the neighborhood N_S of a solution S is defined to consist of moves of a valid task from the current node to an available node.

The tabu search implementation for the grid load balancing problem is graphically depicted in Figure 2. In the initialization phase, a starting solution is created. To this end, a Sufferage algorithm is used to generate the starting solution, and the tabu search algorithm explores the solution space from this starting solution. During the exploration process, the best move is selected, and is then put into a tabu list, to prevent *excessive* 'cycling' of solutions. Moves table are used instead of solutions table due to the impracticality of using solutions table.

Here, the tabu list keeps a record of the movement of a task to a particular computing node. Moves are kept in the tabu list for a period of between TL_{min} and TL_{max} . The exact period for each move is chosen randomly and uniformly from $[TL_{min}, TL_{max}]$. Further to this is an *aspiration condition*, whereby a move that's tabu will be accepted if it improves the best solution found so far.

To complement the exploration process, search *intensification* and search *diversification* are implemented. The inclusion of each type *improves* the performance of tabu search for the load balancing problem.

In search intensification, a modified fitness function is used to decide the next move to be made. In this implementation, moves with a history of good 'score' are rewarded, so that such moves are likely to be chosen during the intensification period. To do this, a 'score' is kept for each move made: the best move for the current iteration that gives a better solution than the solution from the previous iteration is considered good; the difference in value is then recorded. The 'reward' for a move is then the average of the calculated sum. At the start of this intensification period, the tabu list is reset, and the period that moves stay in the tabu list is shortened to $[TL_I, min, TL_I, max]$. The search intensification is triggered when the best solution found is repeated. This intensification procedure would normally last for T_I iterations (unless the *Back to Normal condition* is satisfied).

Here, two types of search diversification procedure are implemented, *mild*, and *hot* diversification. In the

mild diversification procedure, a modified fitness function is again used. Here, frequently made moves are penalized, so that such moves are less likely to be chosen during this diversification procedure. To do this, the ratio of the move to the total moves performed is calculated. The penalty (for a move) is then calculated as the fitness value \times ratio. This procedure is invoked when the search intensification is not able to improve the solution (Figure 2). The procedure lasts for a maximum period of T_{Dm} .

Finally, the hot diversification procedure is global in nature, and is executed when the solution found is not improved within a period T_{stale} . The procedure resets the tabu list, and a new random solution is generated. Further to this, the move counts used in the mild diversification procedure are also reset. In the event that the conditions for search intensification and hot diversification are both satisfied, the search intensification takes precedence. Further, at the beginning of each search intensification and mild diversification, the countdown for T_{stale} is reset.

Each of the search intensification and mild diversification procedure would normally last for T_I and T_{Dm} respectively. However, the procedure will exit and go back to normal exploration if a better solution is found.

Finally, a global stopping condition is used to stop the tabu search. Here, the tabu search is stopped after 200 iterations. An iteration can include a normal exploration, a search intensification, or a search diversification. Table 2 shows the tabu search parameter values used in the experiment:

5 EXPERIMENTS

To analyze the different algorithms presented above, a set of networks and a set of CBoT applications were generated. The algorithms were then applied to the set of network and application pairs. The parameters used for the simulation is given below:

- The generated networks contain computing nodes in the range of [40, 60] nodes with relative processing power in [1, 3]. The communication links exhibit latencies in the range of [0.1, 1] ms with bandwidth in the range of [5, 20] Mb/s. The networks are generated randomly.
- An application contains tasks in the range of [160, 480]. Depending on the processing power of the nodes, each task takes between [100, 1300] seconds to complete.
- Each computing node also has a background workload associated with it. The background workload time

is assumed to follow an exponential distribution with a mean time of 1000 seconds. Discrete background load levels of 0 (no background load), 0.8, and 0.2 are used, with probabilities of 0.5, 0.2, and 0.3 respectively. Such distributions were used as we wanted background workloads that show good variability. Assuming an effective scheduling algorithm, it is variability in the computing nodes' effective processing power that necessitates periodic scheduling, task reassignment, and job migrations.

As the completion time of tasks are dependent on the background workload (which are time variant) present on each computing node, accurate background workload information of each computing node would be ideal. Ideally, the workload information of a node should be sent to the scheduler whenever there is a change; that is, the scheduler has an almost perfect and accurate workload information of all the computing nodes. This however, will result in a lot of messages being sent to the scheduler. On the other hand, the accuracy of the workload information may have an important bearing on the quality of the schedules generated by the load balancing algorithms. Some load balancing algorithms are more sensitive to the accuracies of the workload information, and can generate extremely poor results even when the workload information accuracy is only slightly less than 100%; other load balancing algorithms are less sensitive to such accuracies. For example, in the *First-fit* algorithm tasks are scheduled to the first available computing node, regardless of the speed and background workload of the computing node; in this case the algorithm is insensitive to accuracies of the workload information, as it does not use this information. More sophisticated algorithms however, exploit and use the background workload information to generate better task schedules, albeit with certain sensitivities to the workload information accuracies.

In the experiments below, rather than assuming perfect and accurate workload information, a practical scenario is considered: Each processor sends to the scheduler its background workload information only when it finishes a task. This imperfect information is used for the experiments.

5.1 Results

In the first set of experiment, the genetic algorithm (GA), tabu search (TS), along with Best-fit, Min-min, Max-min, Sufferage, and Random algorithms were applied to ten different networks and applications with the characteristics mentioned earlier. The Random algorithm generates a set number of random task schedules, and chooses the best one – one that gives the minimum makespan. The Random algorithm runs

for the same total combined number of iteration (200 iterations) and population generation ($2 \times$ (size of *TaskSet*)) as the GA. The Random algorithm also considers whole job migrations – the canceling of tasks from one node and reallocating them to another node.

Except for the Best-fit algorithm, the other algorithms considered all have scheduling period of 900 seconds. At each scheduling run, all tasks in the window are considered for scheduling. In this way, schedule can be improved and refined at each scheduling run, and take in to account changes in background workload in processors. At each scheduling run, enough tasks are scheduled on to the processors so that no processor is idle between scheduling run. The results are shown graphically in Figure 3. In the figure, the relative makespan percentages are calculated from the geometric mean of the makespans (as obtained from each algorithm). Geometric mean of the makespans is used instead of the arithmetic mean, as the value of the arithmetic mean can be easily ‘influenced’ by one or two large makespans, essentially dominating and dictating the arithmetic mean. From the graph it can be seen that tabu search (TS) has the lowest makespans, followed closely by genetic algorithm (GA). Best-fit and Random gives the worst results of all the seven algorithms considered. Best-fit does not perform as well because it does not re-schedule tasks once they have been assigned, which is essential with varying background workloads in the computing nodes. Further detailed statistics of the five best performing algorithms are shown in Table 3.

In Table 3, the second column shows the geometric mean of the makespan for each algorithm. *Deviation* represents the average percentage deviation from the best heuristic value. For each algorithm and each simulation run, the deviation value is computed as the relative difference from the best heuristic. The arithmetic mean of all the deviations is then computed. The *Rank* column represents the rank of the values generated from each heuristic. For each simulation run, the best heuristic is given a rank of 1, and the worst given a rank of 7. The arithmetic mean of the ranks is then computed.

The Deviation and Rank columns give an indication of how good each algorithm is. They are both useful, as one algorithm may have a higher rank (closer to one) – the rank is independent of actual makespan values, but also have a higher percentage deviation. For example, Algorithm 1 has rank 1.2 and Algorithm 2 has rank 3; but Algorithm 1 has percentage deviation of 30% and Algorithm 2 has percentage deviation of 10%. Based on the rank Algorithm 1 is preferable to Algorithm 2; however, based on the percentage deviation, Algorithm 2 is preferable to Algorithm 1. That is, Algorithm 2 may rarely give the best performance but is always close to it, whereas Algorithm 1 may give the best performance most often, but is less predictable and

can give really poor results. Depending on the application and usage scenarios, Algorithm 2 may be preferable to Algorithm 1.

From the above we can see TS has a lower geometric mean of the maxspan, much lower average percentage deviation, and higher average rank than the other algorithms. Both the GA and TS has average percentage deviation that is at least two times smaller than the other algorithms. TS also have an average rank that is at least 1 rank above the other algorithms except GA.

5.2 Effect of scheduling period

Periodic scheduling allows prior schedules to be refined and improved with updated system information. This is especially important as there are background workloads associated with the computing nodes; these background workloads – which changes over time, affects tasks' completion time; a higher workload will result in a delay in tasks' completion time, while a change from a high workload to a low workload will result in faster completion of tasks.

In this set of experiment the genetic algorithm (GA), tabu search (TS), Min-min, Max-min, and Sufferage load balancing algorithms are run with different scheduling period. The results are shown in Figure 4. While some of the algorithms show some variability on the results as the scheduling period is increased, there is a general trend of increasing makespans as the scheduling period is increased. As the scheduling period is increased, the load balancing algorithms are less able to re-adjust and re-optimize the task schedules, which is especially important due to the variability of the background workload of the computing nodes. At the extreme end, when scheduling is done only once and no tasks re-scheduling is done, all the algorithms will behave essentially like a purely random algorithm; this result is shown in Figure 5.

6 CONCLUSION

This paper addressed the use of genetic algorithm (GA) and tabu search (TS) to solve the grid load balancing problem. Results of the experiment show that the two methods can be effectively used for grid load balancing. GA and TS shows similar performance results, and performs better than the Best-fit, Random, Min-min, Max-min, and Sufferage algorithms. As the scheduling period is decreased resulting in more scheduling runs, the performance gains from GA and TS increases.

One drawback of the proposed GA and TS algorithms is that they incur extra storage and processing

requirement at the scheduling node. The savings from the proposed GA and TS schemes, however, may outweigh the extra overhead incurred by the schemes, especially considering the ever-decreasing costs of storage and processing power.

Acknowledgement – This work is funded by Smart Internet Technology CRC.
<http://www.smartinternet.com.au>.

REFERENCES

- [1] H. S. Abdinnour and S. W. Hadley, "Tabu search based heuristics for multi-floor facility layout", *International Journal of Production Research*, vol. 38, pp. 365-83, 2000.
- [2] M. A. Abido, "A novel approach to conventional power system stabilizer design using tabu search", *International Journal of Electrical Power & Energy Systems*, vol. 21, pp. 443-54, 1999.
- [3] P. J. Agrell, M. Sun, and A. Stam, "A tabu search multi-criteria decision model for facility location planning", in *Proceedings of the 28th annual meeting Decision Sciences Institute*, 1997, vol. 2, pp. 908-10.
- [4] M. A. S. Al, "Commercial applications of tabu search heuristics", in *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, 1998, vol. 3, pp. 2391-95.
- [5] V. R. Alvarez, E. Crespo, and J. M. Tamarit, "Assigning students to course sections using tabu search", *Annals of Operations Research*, vol. 96, pp. 1-16, 2000.
- [6] G. Barbarosoglu and D. Ozgur, "A tabu search algorithm for the vehicle routing problem", *Computers & Operations Research*, vol. 26, pp. 255-70, 1999.
- [7] R. R. Brooks, S. S. Iyengar, and J. Chen, "Automatic correlation and calibration of noisy sensor readings using elite genetic algorithms", *Artificial Intelligence*, vol. 84, pp. 339-54, 1996.
- [8] I. Forster and C. Kesselman, "The Grid: Blueprint for a New Computing Infrastructure": Morgan Kaufmann, 1998.
- [9] F. Glover, "Future paths for integer programming and links to artificial intelligence", *Computers & Operations Research*, vol. 13, pp. 533-49, 1986.
- [10] F. Glover, "Tabu search. I", *ORSA Journal on Computing*, vol. 1, pp. 190-206, 1989.
- [11] F. Glover, "Tabu search. II", *ORSA Journal on Computing*, vol. 2, pp. 4-32, 1990.

- [12] F. Glover, J. P. Kelly, and M. Laguna, "Genetic algorithms and tabu search: hybrids for optimization", *Computers & Operations Research*, vol. 22, pp. 111-34, 1995.
- [13] F. Glover and M. Laguma, "Tabu search", in *Handbook of Combinatorial Optimization*, vol. 3, D. Du and P. M. Pardalos, Eds. Dordrecht: Kluwer Academic Publishers, 1999, pp. 621-757.
- [14] F. Glover, M. Laguma, and R. Marti, "Fundamentals of scatter search and path relinking", *Control and Cybernetics*, vol. 29, pp. 653-84, 2000.
- [15] F. Glover, E. Taillard, and W. D. de, "A user's guide to tabu search", *Annals of Operations Research*, vol. 41, pp. 3-28, 1993.
- [16] D. E. Goldberg, *Genetic Algorithms in search, optimization and machine learning*. Reading, Massachusetts: Addison-Wesley publishing company inc, 1989.
- [17] K. Hatta, S. Wakabayashi, and T. Koide, "Adapting genetic operators and GA parameters based on elite degree of an individual in a genetic algorithm", *Transactions of the Institute of Electronics, Information and Communication Engineers*, vol. 9, pp. 1135-43, 1999.
- [18] K. Hatta, S. Wakabayashi, and T. Koide, "Adaptation of genetic operators and parameters of a genetic algorithm based on the elite degree of an individual", *Systems and Computers in Japan*, vol. 32, pp. 29-37, 2001.
- [19] A. Hertz and W. D. de, "Using tabu search techniques for graph coloring", *Computing*, vol. 39, pp. 345-51, 1987.
- [20] A. Hertz, E. Taillard, and D. Werra, "Tabu search", in *Local search in combinatorial optimization*, E. Aarts and J. K. Lenstra, Eds. Chichester: John Wiley & Sons Ltd., 1997, pp. 121-36.
- [21] J. H. Holland, *Adaptation in natural and artificial systems*. Ann Arbor: University of Michigan Press, 1975.
- [22] O. H. Ibarra and C. E. Kim, "Heuristic Algorithms for Scheduling Independent Tasks on Nonidentical Processors", *Journal of the ACM*, pp. 280-89, 1977.
- [23] C. L. Karr and L. M. Freeman, *Industrial applications of genetic algorithms*. Boca Raton, FL: CRC Press, 1999.
- [24] H. Kawanishi and M. Hagiwara, "Improved genetic algorithms using inverse-elitism", *Transactions of the Institute of Electrical Engineers of Japan, Part C*, pp. 707-13, 1998.
- [25] C. Kim and H. Kameda, "An algorithm for optimal static load balancing in distributed computer

- systems", *IEEE Transactions on Computers*, vol. 41, pp. 381-84, 1992.
- [26] M. Laumanns, E. Zitzler, and L. Thiele, "On the effects of archiving, elitism, and density based selection in evolutionary multi-objective optimization", in *Proceedings First International Conference Evolutionary Multi Criterion Optimization (EMO 2001)*, 2001, pp. 181-96, Evolutionary Multi-Criterion Optimization. First International Conference, EMO 2001. 7-9 March 2001 Zurich, Switzerland.
- [27] E. D. Lazowska, D. L. Eager, and J. Zahorjan, "The limited performance benefits of migrating active processes for load sharing", *Performance Evaluation Review*, pp. 63-72, 1998.
- [28] H.-C. Lin and C. S. Raghavendra, "A Dynamic Load-Balancing Policy with a Central Job Dispatcher (LBC)", *IEEE Transactions on Software Engineering*, pp. 148-58, 1992.
- [29] M. Maheswaran, S. Ali, H. J. Siegel, D. A. Hensgen, and R. F. Freund, "Dynamic Matching and Scheduling of a Class of Independent Tasks onto Heterogeneous Computing Systems", in *Proceedings 8th Heterogeneous Computing Workshop*, 1999.
- [30] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution programs*. Berlin: Springer-Verlag, 1994.
- [31] M. Mitchell, *An introduction to Genetic Algorithms*. Cambridge, Massachusetts: MIT Press, 1996.
- [32] G. Parks, J. Li, M. Balazs, and I. Miller, "An empirical investigation of elitism in multiobjective genetic algorithms", *Foundations of Computing and Decision Sciences*, vol. 26, pp. 51-74, 2001.
- [33] S. Salleh and A. Y. Zomaya, *Scheduling In Parallel Computing Systems: Fuzzy and Annealing Techniques*. USA: Kluwer Academic Publishers, 1999.
- [34] J. Seijas, C. Morato, and G. J. L. Sanz, "Genetic algorithms: two different elitism operators for stochastic and deterministic applications", in *Proceedings 4th International Conference Parallel Processing and Applied Mathematics (PPAM 2001)*, 2002, pp. 617-25, Proceedings of 4th International Conference on Parallel Processing and Applied Mathematics. 9-12 Sept. 2001 Naleczow, Poland.
- [35] N. G. Shivaratri, P. Krueger, and M. Singhal, "Load Distributing for Locally Distributed Systems", *Computer*, pp. 33-44, 1992.
- [36] R. Subrata and A. Y. Zomaya, "Artificial life techniques for reporting cell planning in mobile computing", in *Proceedings of the Workshop on Biologically Inspired Solutions to Parallel*

Processing Problems (BioSP3), 2002, pp. 203-10, Marriott Marina, Fort Lauderdale, Florida.

- [37] R. Subrata and A. Y. Zomaya, "A Comparison of Three Artificial Life Techniques for Reporting Cell Planning in Mobile Computing", *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, pp. 142-53, 2003.
- [38] M. D. Vose, *The simple Genetic Algorithm: Foundations and theory*. Cambridge, Massachusetts: MIT Press, 1999.
- [39] R. Wolski, N. T. Spring, and J. Hayes, "The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing", *Journal Future Generation Computer Systems*, vol. 15, pp. 757-68, 1998.
- [40] X. Yao, "An empirical study of genetic operators in genetic algorithms", *Microprocessing & Microprogramming*, vol. 38, pp. 707-14, 1993.
- [41] K. K. Yu, "A quasi-static cluster-computing approach for dynamic channel assignment in cellular mobile communication systems", in *Proceedings IEEE VTS 50th Vehicular Technology Conference*, 1999, vol. 4, pp. 2343-7.
- [42] W. Zhu, P. Socko, and B. Kiepuszewski, "Migration impact on load balancing - an experience on Amoeba", *Operating Systems Review*, pp. 43-53, 1997.

Window size w	$8 \times (\text{number of nodes})$
Population m	$2 \times (\text{size of } TaskSet)$
Best cloned σ	5% of m
Tournament selection p_s	0.8
Two-point crossover p_c	0.8
Gene mutation p_m	0.0005
Evolution period T	200
Max stale period T_{stale}	20
Escape mutation p_{mm}	0.2

Table 1: Genetic algorithm parameters.

Tabu period TL_{min}	5
Tabu period TL_{max}	10
Tabu period $TL_{I, min}$	2
Tabu period $TL_{I, max}$	5
Intensification period T_I	13
Diversification period T_{Dm}	13
Max stale period T_{stale}	20

Table 2: Tabu search parameters.

Algorithm	Geometric mean	Deviation (%)	Rank
GA	4646	4.37	2.4
TS	4526	1.48	1.8
Min-min	5136	15.54	3.8
Max-min	4904	10.19	3
Sufferage	5136	15.47	4

Table 3: Statistics for the different algorithms.

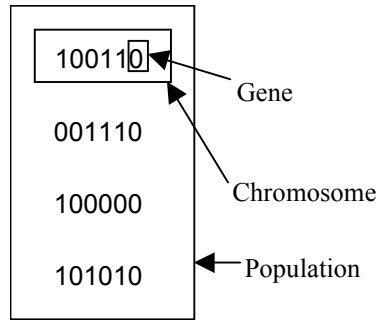


Figure 1: Genetic algorithm system. Population is a collection of chromosomes, and chromosome is a collection of genes.

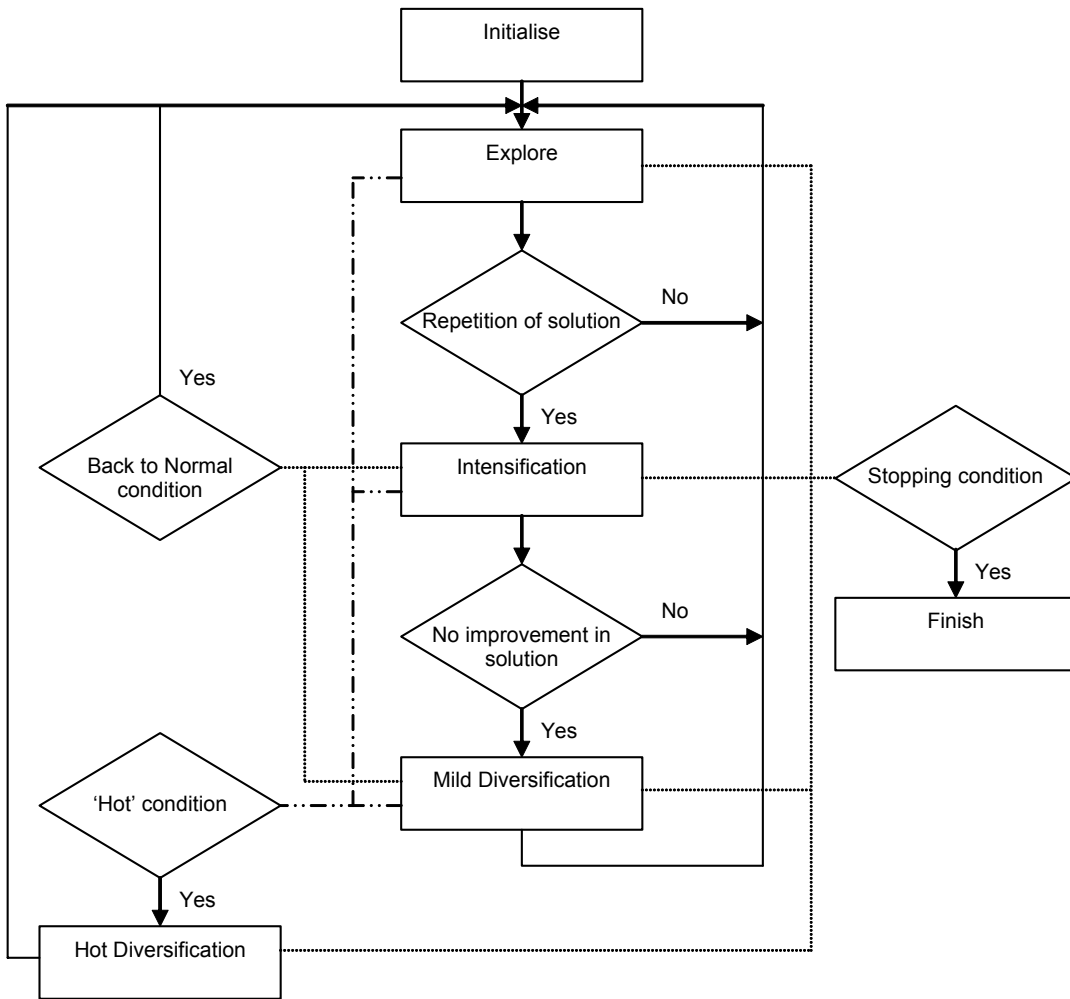


Figure 2: Tabu search procedure.

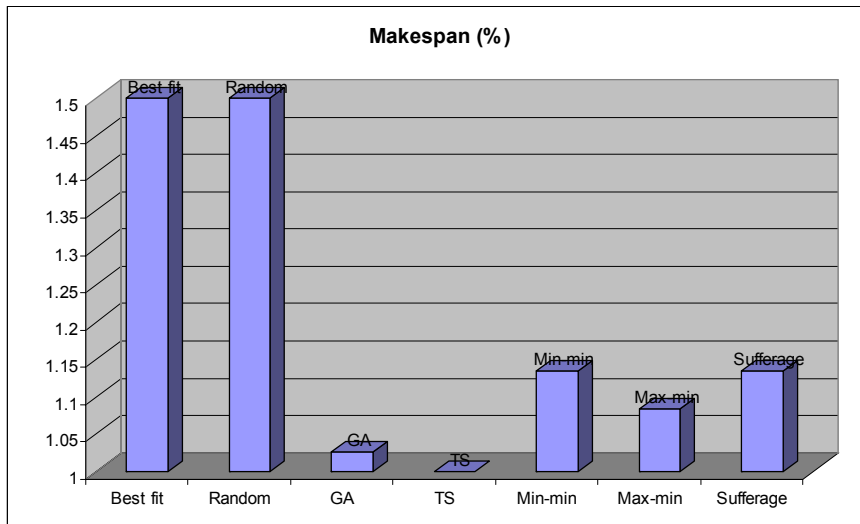


Figure 3: Results for the different algorithms.

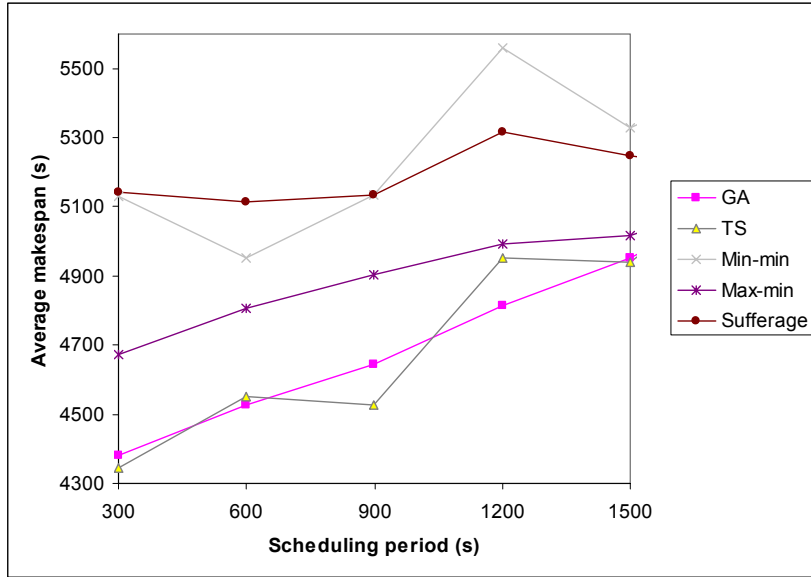


Figure 4: Effect of scheduling period on the geometric mean of the makespans.

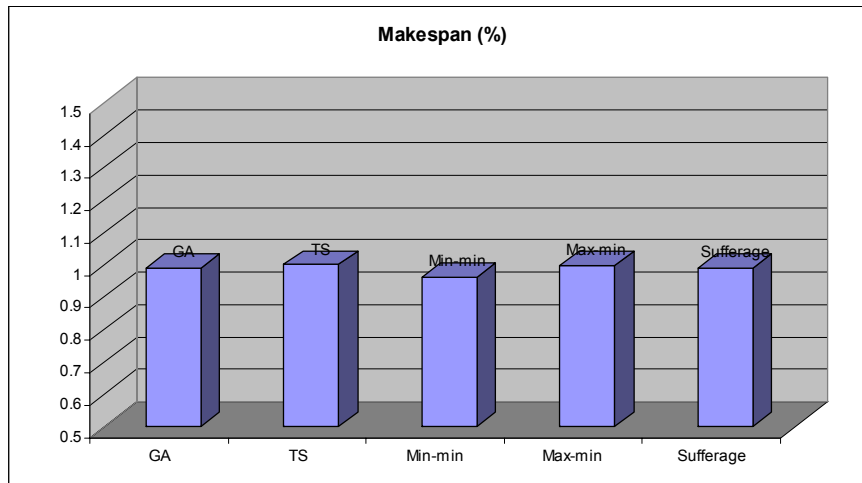


Figure 5: Geometric mean of makespans with no periodic scheduling.