Neural Computing and Applications (2023) 35:18035–18057
https://doi.org/10.1007/s00521-023-08682-y

ORIGINAL ARTICLE

An efficient lightweight algorithm for scheduling tasks onto dynamically reconfigurable hardware using graph-oriented simulated annealing

Morteza Mollajafari[1]

Received: 7 August 2021 / Accepted: 10 May 2023 / Published online: 1 June 2023

Abstract
Scheduling complex applications as task graphs on finite computational resources assuring task interdependencies is a well-known NP-complete optimization problem. This problem is well-addressed for microprocessor systems but for Dynamically Reconfigurable Hardware (DRHW) systems in which, in addition to tasks, the reconfiguration time and complexity also have to be scheduled; this problem is more complicated. DRHW reconfiguration overhead is considerable and can be crucial for real-world applications. To deal with this overhead, in this paper, a meta-heuristic method named Graph-Oriented Simulated Annealing (GOSA) is proposed. By introducing an innovative graph and solution structures called schedule graphs, and also some controlling functions which are inherited from the nature of the problem, the proposed method adapts itself to the characteristics of the problem. This helps the algorithm to adjust its exploration and exploitation speed and accuracy according to the requirements of the given problem and consequently find high-quality solutions quickly. To demonstrate the performance of the proposed method, it was tested on several synthetic and real-world benchmark task graphs, and the results were compared with a selection of classic and state-of-the-art algorithms. The method is comprehensively evaluated by performing numerous experiments in terms of execution time, makespan, scalability, and reliability. The results of the experiments on benchmarks show that in terms of the quality of the solutions, GOSA outperforms BGA, HPSO-GA, and FATS by 17%, 13%, and 5%, respectively, and its execution time is considerably less than all competing algorithms. Moreover, according to the experiments done on synthetic graphs, the makespan of the solutions generated by GOSA, Genetic Algorithm (GA), and the Gxhaustive Search over the List Scheduler are improved on average by 7.2%, 8.1%, and 19.1%, respectively. The most significant achievement of the proposed method is its execution time which is 31 times faster than GA. Finally, the results confirm that the proposed method is scalable for large task graphs, and its reliability is superior.

Keywords Dynamically reconfigurable hardware · Genetic algorithm · Graph-oriented · Makespan · Simulated annealing · Task scheduling

1 Introduction

Dynamic reconfiguration is an intelligent technology since its computation speed is similar to that of custom hardware, and its flexibility is comparable to that of a general-purpose processor [1, 2]. Dynamically Reconfigurable Hardware (DRHW) systems, which are now realistic and commercially available, are died in which embedded microprocessors, on-chip memory, and reconfigurable logic blocks are integrated. Such systems assure the flexibility of traditional general-purpose processors and provide the efficiency and high performance of Application-Specific Integrated Circuits (ASICs). To well benefit from the advantages of these systems, employing a proper scheduling algorithm is necessary. Hence, in this paper, we try to develop an efficient scheduling algorithm for DRHW systems.

✉ Morteza Mollajafari
mollajafari@iust.ac.ir

[1] Vehicle Electrical and Electronic Research Lab, School of Automotive Engineering, Iran University of Science and Technology, Tehran 16846-13114, Iran

The objective of DRHWs scheduling problem is comparable to that of the classic multiprocessor scheduling since both are trying to assign some processors to a set of tasks so that they minimize the overall makespan. However, due to the existence of run-time reconfiguration, which is a prerequisite for executing a task in a hardware form, the DRHW task scheduling is more complicated. Here, in contrast with multiprocessor scheduling; besides the task scheduling, the task allocation in configuration SRAM must be taken into consideration. If the configuration schedule is not done correctly in coordination with task schedules, the resulting latencies will also affect the execution of the scheduled tasks. This can vastly decrease the efficiency of the system. It has been proven that this procedure, called DRHW task scheduling, is an NP-complete problem [3, 4]. On the other hand, if we want the algorithm to be practicable for real large applications, its execution time must be short enough and has a low order of complexity.

To tackle the mentioned challenges, some fast task scheduling algorithms must be developed to minimize the latencies incurred by task configuration procedures and fully utilize the periods during task executions for configuring subsequent tasks. To solve the task scheduling problem for parallel computing systems like reconfigurable hardware systems, plenty of work has been carried out [5–9]. Among these works, nature-inspired and evolutionary algorithms are at the center of attention. The reason behind this is that according to the above complexities, the search space of this problem becomes multi-dimensional and huge. Hence, solutions based on Exhaustive Search are not feasible as the operating cost of generating schedules is very high. Therefore, there are no heuristics that may produce an optimal solution within the polynomial time for such kinds of problems. Generally, two optimization approaches can be exploited to solve this problem: deterministic and stochastic. Deterministic methods in both gradient-based and non-gradient-based groups are effective in linear, convex, uncomplicated, low-dimensional, and differentiable problems, but lose their effectiveness in dealing with optimization problems like ours that have features such as complex, high-dimensional, and discrete search space. These reasons lead us to use stochastic approaches, i.e., meta-heuristic optimization algorithms that apply random operators, random search, and trial-and-error processes. In the problems like task scheduling onto Dynamically Reconfigurable Hardware, it is preferable to find a suboptimal solution (rather than optimum points) in a short time. Meta-heuristic-based techniques have been proven to achieve near-optimal solutions within a reasonable time for such problems [10]. This is the main reason to use these techniques for our problem. Moreover, developing a heuristic that is customized for the problem could efficiently solve the scheduling problem in a reasonable amount of time which is not trivial. However, the meta-heuristic methods can address different forms of the problem. Therefore, to solve the scheduling problem efficiently, a meta-heuristic approach must be adopted and adapted which is efficient, lightweight, fast, scalable, and applicable.

The nature of random search in meta-heuristic algorithms leads to the fact that there is no guarantee that this best candidate solution is the best solution (known as the global optimal) to a problem. Therefore, the best candidate solution is known as a quasi-optimal solution, which is an acceptable solution and close to the global optimal. According to the No Free Lunch (NFL) theorem [11], any two optimization algorithms are equivalent when their performance is averaged across all possible problems; and it also states that an algorithm may have a successful implementation on some optimization issues but fail to address others. Therefore, no one can claim which algorithm is generally more suitable for finding the quasi-optimal solution for a specific problem. Instead, an algorithm should be selected which is more compatible with the nature and the search space of the problem. Moreover, efforts should be made to fit and customize the algorithm based on the characteristics and requirements of the problem. There are too many meta-heuristic methods that can be exploited to solve the problem of task scheduling in parallel computing systems [12]. Among, those will be successful on a problem, which can provide a balance between the exploration (that is well investigating the whole search space) and the exploitation (that is identifying those parts of the solution space with high-quality solutions and intensify to search them) in the problem solution space. Most of these algorithms have two main weaknesses: having multiple parameters which interact with each other and, in turn, make their optimal tuning a tedious task and having high-time complexity which makes them less practicable for real and time-critical applications [13]. Moreover, by increasing the application size, these algorithms miss their consistency, and hence, the quality of their solutions would be significantly reduced [12]. In other words, the algorithms are not scalable as the problem gets bigger. Another essential factor of the algorithms is their applicability which means maintaining efficiency despite varying the characteristics of the target system. Although some algorithms provide remarkable solutions for the problem, since they are designed and adapted for a particular type of target system, by changing the system specifications, they probably miss their performance.

To produce high-quality solutions while strongly decreasing the algorithm run-time, and simultaneously filling the mentioned gaps, in this paper, the Simulated Annealing (SA) algorithm is adopted and adapted based on the nature and requirements of the reconfigurable hardware task scheduling problem. As far as known to the author, up to the present time, no study has been carried out on the application of SA to solving task scheduling problems on generic DRHW systems, even in its general and basic form. To solve the problem, we developed a modified and customized SA called Graph-Oriented Simulated Annealing (GOSA), in which some novel controlling functions inherited from the nature of the problem are introduced to the algorithm. These functions control neighborhood zone selection and the algorithm's step lengths based on a new parameter named height value. Moreover, an innovative graph and solution structure called schedule graph (S-graph), which is fitted to the profiles of the reconfigurable system's task scheduling problem, is introduced. This graph guarantees the precedence constraints and system limitations so that the algorithm will be practicable for multiple target systems with different types and features as well as for various applications with different graph structures.

The main contributions of the paper can be summarized as follows:

- Application of the SA algorithm to solving task graph (DAG-type) scheduling problem on a generic parallel reconfiguration model of DRHWs for the first time.
- Presenting a novel solution representation scheme that is based on two-dimensional strings.
- Introducing a new graph called S-graph which implicitly guarantees the precedence constraints and the target system limitations.
- Proposing a new neighborhood selection mechanism adapted to the characteristics of the S-graph.
- Presenting an innovative cooling function customized to the problem search space size.
- Applying three inventive mutation-based operators for generating new solutions in each stage.

In the following, the literature which addressed solving the DRHW task scheduling problem is reviewed with a focus on the evolutionary-based meta-heuristic ones [14–19]. The advantages and disadvantages of each work are determined, and their gaps for future works are discussed.

GA is one of the popular, efficient, and well-known tools for solving our problem [19–22]. Correa et al. have improved GA for solving the problem by using the List heuristic in GA operators, so that it improves the quality of the solutions and slows down the run-time of the algorithm in comparison with those of standard GA and List Scheduler. In another work [23], the authors tried to combine two evolutionary algorithms (GA and Particle Swarm Optimization (PSO)), to solve the multiprocessor scheduling problem. Although the obtained quality is much better than the individual algorithms, it is clear that its convergence time is too much longer than those of others due to the existence of two consecutive population-based algorithms in their approach.

In [24], Bonyadi et al. proposed a scheduling algorithm for RC systems called Bipartite GA (BGA), which uses different genetic operators and chromosome representations for each part of the algorithm. In the first phase of this two-phase algorithm, the proper order of tasks is found by a GA, and in the next phase, another GA finds the most suitable processors for executing each task. The results of simulation results show that their approach needs 10% fewer iterations compared to its competitor algorithms on average. However, the volume of computations required for a generation might vary from one algorithm to another, and it means that the algorithm with more iteration numbers has much less execution time in comparison with another algorithm.

In [25], an Ant Colony Optimization (ACO) algorithm-based method known as Feasibility Assured TSP-likened Scheduling (FATS) is presented. This work adapts the ACO to the specifications of the scheduling problem by converting the task and resource graphs into a construction graph. Then, an ACO is used to solve it, similar to the Travelling Sales Person (TSP) problem. The main feature of this graph is that it preserves the tasks' precedence constraints and target system restrictions in its structure. Although the solutions provided by the ACO are always feasible and have high qualities, its run-time is too much due to its population nature.

A discreet Invasive Weed Optimization (IWO) algorithm combined with the Earliest Finish Time (EFT) approach is used in [26]. The latter method assigns a priority to each task while the former performs the task to processing resources mapping. Although the results show high-quality solutions of this algorithm, it still suffers from the last longing execution time due to its two-step nature. Moreover, it may lose its performance when the configuration time overhead is added to the problem.

Some efforts also have been done exploiting SA. In [27], an SA method is presented, which decreases the optimization procedure run-time and makes the algorithm scalable to the application size. In [28, 29], the authors investigated the usage of SA for solving task mapping and scheduling problems. Several experiments on SoCs confirm that the solutions generated by SA have acceptable and considerable high-quality makespan, while the run-time is remarkably much less.

In another work, a hybrid modified SA and List heuristic is presented [30], which has two stages. In the first stage, the List heuristic provides some early solutions, and then in the next stage, these solutions are improved by using a modified SA. By performing some simulations, they showed that their approach overtakes the List-based scheduling algorithm regarding the quality of the solutions.

The authors in [31] presented a modified ACO scheduling algorithm that applies special constraints for generating new solutions and designing particular pheromone tables. From the quality point of view, their results demonstrate that the solutions produced by their proposed ACO are 11% better than those of SA and TS on average [6, 32].

As can be inferred from the above survey, GA is a popular and successful technique that offers short makespan solutions but with more algorithm run-time. Many pieces of the literature have addressed the usage of GA for solving this problem, each has considered only one or two of the mentioned metrics in the previous section, and none carried out experiments to investigate all of them. In this regard, we tried to enrich the SA by customizing it to our problem.

The rest of the paper is organized as follows: Sect. 2 presents the task and device models used in this study and the details of the proposed method. The results of the experiments are presented in Sect. 3. In Sect. 4, a discussion of the proposed method is provided. Finally, the paper is concluded in Sect. 5.

2 The proposed method

By the scheduling problem in this paper, one means to minimize the overall makespan of a set of dependent tasks on a given DRHW system by providing a feasible and effective task schedule. Here, in contrast with multiprocessor scheduling, besides task scheduling, task allocation is taken into consideration. In addition, we must carefully treat the configuration prefetching due to the existence of run-time reconfiguration.

2.1 Device model

Among different types of DRHW, in this work, a parallel reconfiguration model is used as the target model. The key reason for using this model is its high degree of generality, which allows describing different types of reconfigurable systems by intruding different values for the model's parameters.

The parallel reconfiguration model, which is a part of SoC platforms, consists of several continuously connected identical tiles. As depicted in Fig. 1, each tile consists of two elements, a logic circuit and its configuration controller. Any set of m consecutive connected tiles can accommodate a task requiring m resources. The crossbar connection is provided to connect the configuration SRAMs of the processing resources to parallel configuration controllers [33].

2.2 Task model

Typically parallel programs are composed of some dependent tasks, which are usually modeled by directed acyclic graph (DAG), $G(V,E)$, where $V = \{j_1, j_2, \ldots, j_n\}$ is a finite set of nodes or tasks, and E is a set of directed edges between the tasks that denote the task dependencies in the form $(j_i, j_k) \in E$, where j_i is the parent of j_k, and the data produced by j_i will be used by $\mathrm{j_k}$. A child task cannot be executed until all of its predecessors are accomplished. The number of requisite tiles, R_i, and execution time, RT_i, are two attributes assigned to each task i. Since configurations of each task are explicit processes but do not being presented in the task graph, ordinary DAG representation is not sufficient. For this study, an extended graph named $G^+(V^+, E^+)$ is used instead. Additional nodes are added by the symbol V', which are representative of the configuration nodes, and each node of this type represents the configuration of one tile. Also, additional links named E' are added from V' to V since configurations must be done prior to execution. In short, to describe a mathematical relationship between the original and the comprehensive DAG, one can notate it as $V^+ = V \cup V'$ and $N^+ = N \cup N'$. In this regard, an original DAG with five nodes or tasks and its associated extended DAG with 16 nodes are shown in Fig. 2a and b, respectively. The configuration node representing the configuration of the j th segment of task i is represented by $C(i,j)$ [35].

2.3 Problem formulation

The problem in this study is providing solutions for scheduling the DAG tasks on hardware tiles beside their configurations so that the time of running the tasks is minimized. Each DAG which denotes by $G(V,E)$ consists of two attributes, $V = \{j_1, j_2, \ldots, j_n\}$, a finite set of nodes or tasks, and E, a set of directed edges between the tasks. Therefore, there are n tasks in a DAG which are shown by the set V. To calculate the makespan of running a DAG on a DRHW, two attributes must be considered for each task; the number of requisite tiles, R_i, and execution time, RT_i, for the task T_i. If the task T_i is mapped on m tiles or R_j resources $(Map_{i,j} \equiv (T_i \rightarrow R_j))$, the time of occupation of the T_i on the DRHW includes the time of its configuration (CtT_i) on the system and its running time on the allocated tile(s). This parameter can be calculated by Eq. (1).

$$OT_{T_i|Map_{i,j}} = \frac{R_i \times RT_i}{n_{R_j}} \tag{1}$$

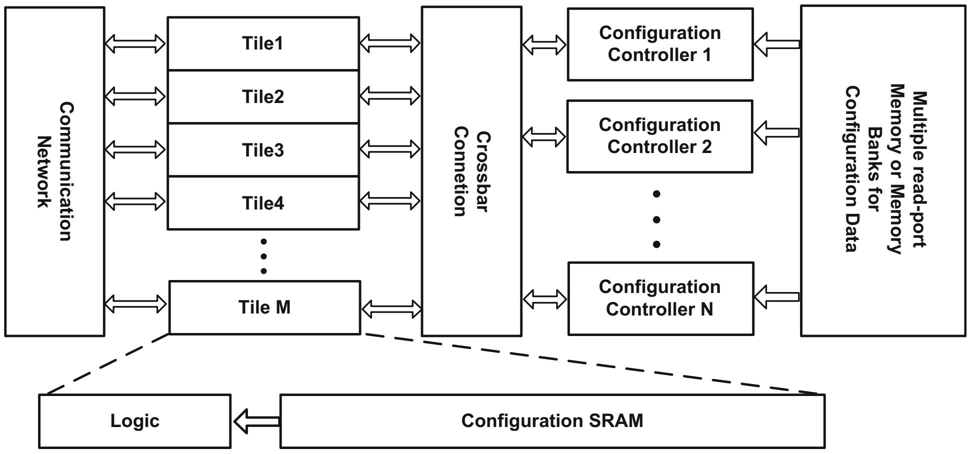

Communication Network
Tile1
Tile2
Tile3
Tile4
Tile M
Crossbar Connetion
Configuration Controller 1
Configuration Controller 2
Configuration Controller N
Multiple read-port Memory or Memory Banks for Configuration Data
Logic
Configuration SRAM

Fig. 1 A generic parallel reconfiguration model [34]

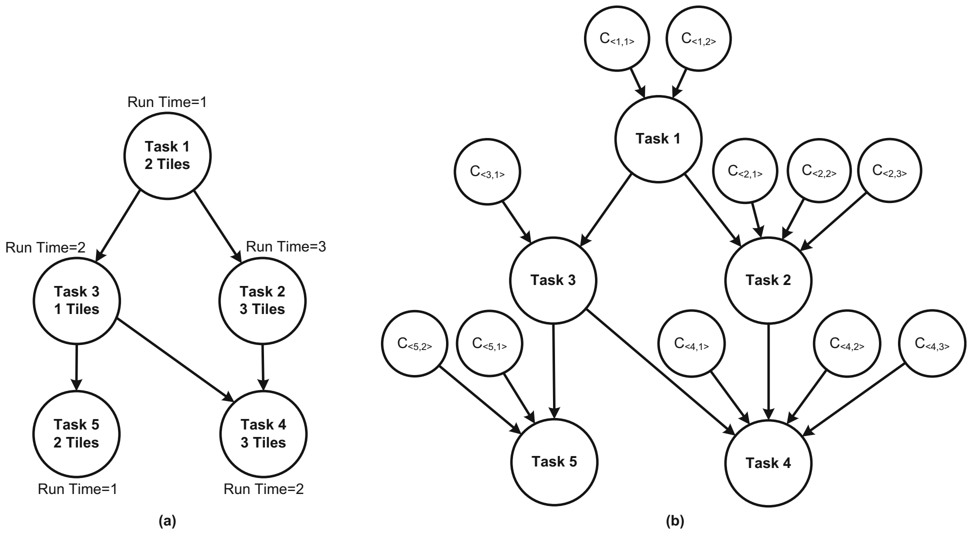


Fig. 2 **a** A normal or original DAG with five tasks and **b** its associated extended DAG with 11 tasks

$\mathrm{ET}_{\mathrm{T_i|Map_{i,j}}} = \frac{\mathrm{IN_{T_i}}}{\mathrm{Speed_{R_i}}}$ where n_{R_j} indicates the number of assigned tiles for executing the task R_j, and $OT_{T_i|Map_{i,j}}$ denotes the occupation time of task T_i on the DRHW according to the assigned n_{R_j} tiles. Equation (1) gives the occupation time of each task individually but not its completion time from the beginning of the execution of the whole DAG. To compute the completion time of a task T_i, the overall time needed to complete the execution of a task T_i, and all its serially executed predecessors are required. According to Eq. (1), the completion time of the task T_i from the moment of beginning the first task in the DAG can be calculated as Eq. (2).

$$CT_{T_i} = \begin{cases} OT_{T_i|Map_{i,j}} & \text{if } Pred(T_i) = \emptyset; \\ \max\limits_{T_k \in Pred(T_i)} \left\{ OT_{T_i|Map_{i,j}} + CT_{T_k} \right\} & \text{otherwise;} \end{cases} \tag{2}$$

where $Pred(T_i)$ denotes the predecessor tasks of T_i, which means that those tasks exist before T_i in the task graph. Therefore, the total time of the DAG execution on a given DRHW (i.e., the makespan of the DAG), W, is the maximum value resulting from the above equation computed for all DAG tasks. This makespan can be calculated as Eq. (3).

$$ET_W = \max_{1 \le i \le n, T_i \in V} CT_{T_i} \tag{3}$$

Equation (3) denotes that the execution time or makespan of a DAG equals the completion time of its most time-consuming task according to all its predecessor tasks. Therefore, to calculate the ET_W, the CT_{T_i} of all the graph's tasks must be computed first, and then, the largest will be picked.

2.4 The proposed algorithm

In the proposed method, SA is used as the core algorithm. SA is a probabilistic non-greedy algorithm, which is inspired by the process of melting and cooling materials. It searches the solution space of a problem by annealing from a high to a low temperature. The annealing process, which is controlled by a cooling function, must be designed so that in high temperatures, the algorithm explores the solution space to escape from local minimums and reach into the global valley. Then, by decreasing the temperature gradually, it intensifies its investigations to find the sub-optimal solution.

SA has many advantages, some of which are: Any arbitrary objective function or system can be handled by this algorithm; theoretically or to some extent assures reaching into the global minimum solution and usually provides high-quality and near-optimal solutions; it is reasonably not hard to implement; finally, it deals with a single result or individual and requires a little computation to go from a stage to another. Because of working on a single solution and its concise steps from one iteration into another, the main drawback of this meta-heuristic algorithm is its slow convergence. One can tackle this issue by adjusting the steps' lengths according to the problem and iteration number so that, on the one hand, the mentioned balance would not be disrupted. On the other hand, the algorithm reaches the global minimum in a guided random search manner. In this regard, we have made the algorithm's exploration intelligent by proposing a customized neighborhood selection mechanism and a problem-dependent cooling function. Moreover, we have exploited innovative mutation operators to intensify the algorithm's exploitation in promising areas.

To use SA for solving the DRHW scheduling problem, first, it is necessary to modify and adapt the algorithm's controlling parameters and operators. These should be done to cover its weaknesses resulting from its single-solution searching nature while exploiting its light-computational complexity as well. In other words, one must customize it to the reconfigurable hardware scheduling problem to well explore its huge multi-dimensional search space in a reasonable time. This requires some mechanisms for adjusting the search steps, including the search speed and zone based on the problem size and specifications. To achieve the mentioned goals, we developed innovative problem-dependent controlling functions. First, the cooling function has been customized for the DRHW problem size. In this regard, the cooling steps must differ for problems with different sizes. The rationale behind it is that problems with small sizes need fewer steps to search the whole solution space. However, the large ones require much more iterations to fully explore different search space zones. To realize such a dynamic cooling function, we calculate the temperature of every iteration based on the number of task graph nodes, $|V|$, and a coefficient that will be further tuned to find the optimal relation. Another function that is adapted to the problem specifications is neighborhood selection. To this end, we introduced some mutation operators as well as a new parameter named height value which shrinks the searching neighborhood of the current potential solution gradually and according to the current temperature. In the following, the details of these functions are presented. The flowchart of the GOSA is shown in Fig. 3. At first, the algorithm's parameters and the necessary functions are set and defined. Before starting the algorithm, we must define a representation scheme for the potential solutions.

In our method, solutions are represented using pairs of two-dimensional strings. The first two-dimensional string, called task string (*T-string*), represents the scheduling

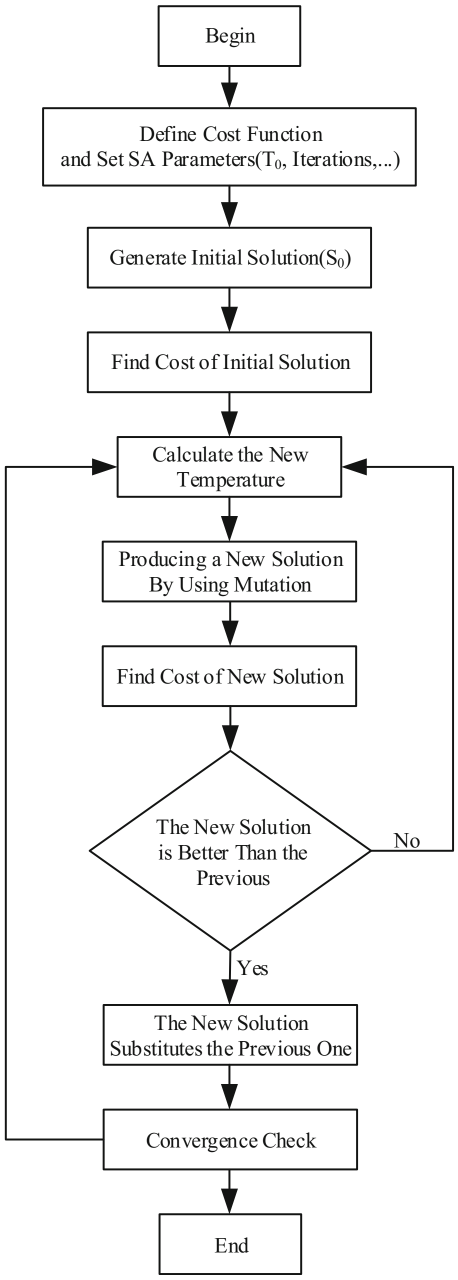

Begin
Define Cost Function and Set SA Parameters(T_0, Iterations,...)
Generate Initial Solution(S_0)
Find Cost of Initial Solution
Calculate the New Temperature
Producing a New Solution By Using Mutation
Find Cost of New Solution
The New Solution is Better Than the Previous
No
Yes
The New Solution Substitutes the Previous One
Convergence Check
End

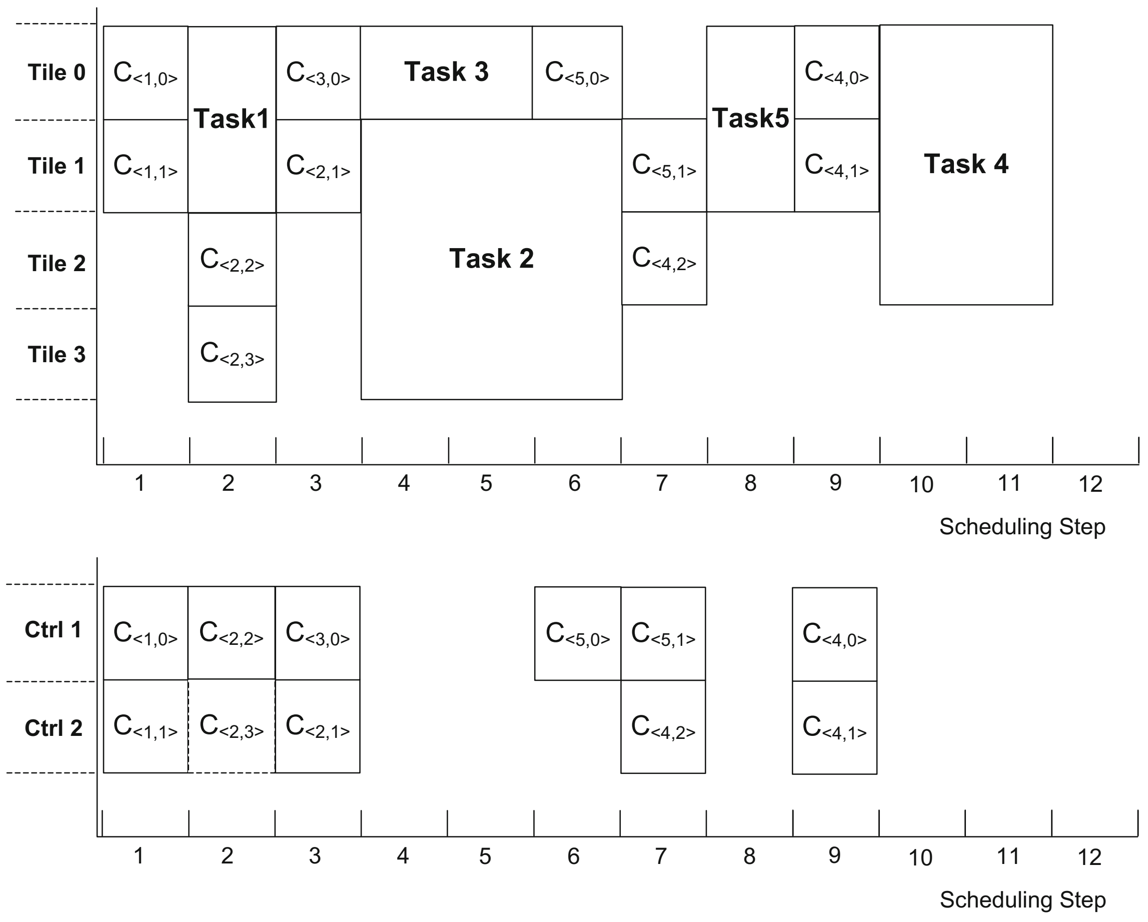
Fig. 3 Flowchart of the GOSA

results of tasks on tiles $\{Tile_1, Tile_2, \ldots, Tile_n\}$. Each tile includes a set of tasks that are scheduled in it, and the execution order of the tasks is denoted by their positions in the tile. If a task needs more than one tile, it appears in all tiles assigned to it.

The second two-dimensional string, controller string (*C-string*), represents the configuration scheduling results $\{Ctrl_1, Ctrl_2, \ldots, Ctrl_n\}$. Each controller includes a set of configurations that are scheduled on it. The position of each controller in the string shows the order of using that controller. Figure 4 depicts an example of two-dimensional strings and their scheduling for the extended graph in Fig. 2.

Now, we can generate an initial scheduling solution, S_0. This solution is produced through a resource-constraint list scheduling approach in which the resources are selected accidentally. The basic procedure for creating the S_0 is as follows:

Pace (1): A ready task node is selected. A task node is ready when all of its preceding task nodes, if any, are scheduled.



Task Strings

Tile 0: (task1,task3,task5,task4)

Tile 1: (task1,task2,task5,task4)

Tile 2: (task2,task5,task4)

Tile 3: (task2,task4)

Controller Strings

Ctrl1: ($\mathbf{C_{<1,0>}}$,$\mathbf{C_{<2,2>}}$,$\mathbf{C_{<3,0>}}$,$\mathbf{C_{<5,0>}}$,$\mathbf{C_{<4,2>}}$,$\mathbf{C_{<4,1>}}$)

Ctrl2: ($\mathbf{C_{<1,1>}}$,$\mathbf{C_{<2,3>}}$,$\mathbf{C_{<2,1>}}$,$\mathbf{C_{<5,1>}}$,$\mathbf{C_{<4,0>}}$)

Fig. 4 Results of task and configuration scheduling

Pace (2): Controllers for the task node configuration node(s) and tile(s) for the task node are randomly selected. If more than one tile is needed, consecutive joint tiles are randomly selected.

Pace (3): If any task nodes are left unscheduled, go back to pace (1); otherwise, the initial solution is already produced, and the procedure is completed.

After the initial solution is produced, its cost is calculated using an objective function computing the makespan of the scheduling solution as in [36]. Then, the cooling function calculates the new temperature as a function of some parameters, which will be discussed later.

Now, the algorithm starts to search the solution space iteratively based on the current solution. For generating new solutions, a new graph called a scheduling graph (*S-graph*) originated from the strings' pairs of the current solution by realizing additional links of the scheduling dependencies in the comprehensive graph. On each tile of *T-strings*, the extra edge(s) is raised from the node at the position P_{th} to the configuration node(s) of the node at position $(P+1)_{th}$ to ensure the correct order of execution. For example, edges are inserted from task 3 to configuration nodes $C_{\langle 5,1\rangle}$ and $C_{\langle 5,2\rangle}$ of Fig. 2. On each controller of *C-strings*, the extra edge(s) from the configuration node(s) at position N_{th} to the configuration node(s) of the node at position $(P+1)_{th}$ are inserted into the extended DAG. For example, since $C_{\langle 2,3\rangle}$ should precede $C_{\langle 4,2\rangle}$, an extra edge is inserted from $C_{\langle 2,3\rangle}$ to $C_{\langle 4,2\rangle}$. Therefore, it is guaranteed that every phase of generating new solutions does not break priority rules. By modifying the current solution's elements randomly, new solutions are achieved. To this end, the mutation operator of GA is employed. As the task and configuration nodes are two different types of nodes, they can be mutated independently. We used three kinds of mutations in our method. In the first one, only *T-strings* are mutated by randomly selecting a task node and taking it to a new place. This changing of a task node location should meet the condition of Eq. (4) so that the precedence conditions are preserved:

$$\begin{aligned}&height\ (the\ node\ before\ V_i)\\&<height(V_i)\leq height\ (the\ node\ after\ V_i)\end{aligned}\tag{4}$$

where $height(V_i)$ is the height value of the node V_i. The $height(V_i)$ is gained according to the *S-graph* as relation (5):

$$height(V_i)=\begin{cases}1, & if\ V_i\ is\ a\ root\ node\\ 1+\max(height\ (predecessors)), & else\end{cases}\tag{5}$$

In the second mutation, only *C-strings* are changed by accidentally selecting a configuration node and taking it to a new controller's corresponding *C-string*. The conditions of the new location are like that of task mutation, but the height value of each configuration here is the same height value of the task node it configures.

The third mutation rotates the assignment of controllers to the configuration nodes of a task. This mutation randomly selects a task node, T_i and applies only to tasks that need more than one tile. If L tiles are required for the task, i.e., it has L configuration nodes; node $C_{\langle i,1\rangle}$ is substituted by $C_{\langle i,2\rangle}$, $C_{\langle i,2\rangle}$ by $C_{\langle i,3\rangle}$, until $C_{\langle i,1\rangle}$ substitutes $C_{\langle i,N\rangle}$.

There is a certain probability for each of these mutations to run while generating a new solution. This way, the task nodes in *T-strings* are ensured to keep the correct order according to their height values, but the *C-strings* might disturb the order and invert the height values. This inversion may make a cycle in the *S-graph* after the random modification of scheduling dependencies done in mutation. Therefore, to guarantee the feasibility of new solutions, the *C-strings* must be sorted at the end of the mutation operation, according to the height values.

The run probability of each mutation mentioned is determined by:

$$Random(0,1)<T\tag{6}$$

where $Random(0,1)$ is a random number generator between 0 and 1, and T is the temperature of the current iteration. This function runs in every phase independently

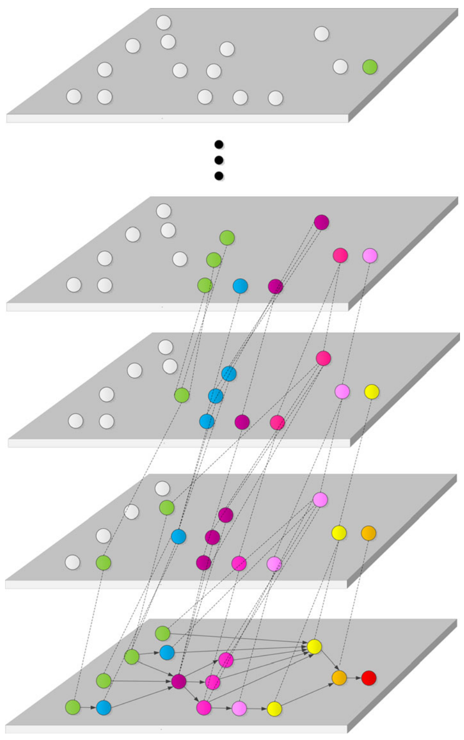


Fig. 5 The proposed algorithm's procedure scheme for selecting mutant nodes as the algorithm progresses for the example DAG

for each of the three kinds of mutation; if the mentioned condition is met, the mutation applies to the solution string. It should be noted that all three kinds of mutation may apply in one iteration.

During the algorithm's progress, the temperature decreases (falls from 1 to 0). This is called a cooling plan. Various cooling plans exist, such as linearly decreasing, geometrically decreasing, Hayjek optimal, etc. The cooling schedule in this study, as mentioned before, depends on the size of the problem and is as follows:

$$T = \frac{MI - CI}{MI} \tag{7}$$

where MI is the maximum number of iterations, and CI is the current iteration number. The number of iterations to reach a suboptimal solution is directly proportional to the number of tasks, $MI \propto N_{task}$ (N_{task} is the number of tasks). A coefficient, k, converts the proportion to Eq. (8):

$$Maximum\ Iterations = k \times N_{task} \tag{8}$$

So, Eq. (7) becomes as relation (9):

$$T = \frac{k \times N_{task} - CI}{k \times N_{task}} \tag{9}$$

In the beginning, when the temperature is high, the algorithm is not much greedy and may generate a solution different from the current solution. But as time passes, the temperature falls, and the algorithm gradually becomes greedier. Therefore, the new solutions do not differ much from previous ones, and the solutions with less fitness are less likely to be accepted. In our proposed method, in the early stages of the algorithm, when the temperature is high, the mutation can be applied to more nodes of the solution string. As a result, the solution space is explored thoroughly, and the algorithm does not get stuck in a local minimum. As the temperature decreases, the number of nodes of the solution string mutated decreases so that new solutions occur in the neighboring of the previous one. Gradually, by exploring the global minimum valley, the global minimum can be reached in the final stages. Two out of three mutations discussed before are dependent on height values. The height value of the selected node for mutation should satisfy the following condition:

$$height \in (1 + (1 - T) \times (height_{max} - 1), height_{max}) \tag{10}$$

where $height_{max}$ is the maximum height value of nodes. Therefore, as time passes and T decreases, the limits of the neighborhood range in which nodes are selected decrease from $(1, height_{max})$ to only $height_{max}$. The proposed algorithm's procedure scheme for selecting mutant nodes as the algorithm progresses for the example DAG is presented in Fig. 5.

In each iteration, the fitness of each new solution is measured as in [36], in inverse proportion to the scheduling length. If this fitness is better than the previous, the new solution will be substituted; otherwise, it will be accepted with the following probability:

$$P_{accept} = \begin{cases} 1; & \Delta Cost \leq 0 \\ e^{-\frac{\Delta Cost}{T}}; & \Delta Cost \leq 0 \end{cases} \tag{11}$$

where $\Delta Cost$ denotes the difference between the cost of the generated solution and the previous one, and T is the temperature of the current iteration.

At the end of each iteration, the stopping criteria of the algorithm are checked, and the algorithm exits if one of them is met. Otherwise, it iterates from the first stage. As an example, the results of scheduling the DAG of Fig. 2 on a DRHW system with $NT = 3$ and $NC = 2$ by using the proposed algorithm are shown in Fig. 4. The steps of GOSA are presented in detail in Algorithm 1.

Algorithm 1. Graph-Oriented Simulated Annealing

```
1:  start procedure GOSA
2:    Input: DAG ($G+$), $NC$ (Number of Configuration Controllers), $NT$ (Number of Tiles)
3:      $CI$ (Current Iteration) = 0
4:      $MI$ (Maximum Iterations) = k.$N_{Tasks}$
5:      $T$ (Temperature) = 1
6:      initialize ($S_0$ (initial task scheduling), $C_0$ (initial cost))
7:       for $i$ = 1 to $N_{Tasks}$ do
8:         $t_r$ = a ready task (a task node which all of its preceding tasks are scheduled) selected randomly
9:            select free controller(s) for $t_r$'s configuration node(s)
10:           select free (consecutive) tile(s) for $t_r$ node
11:      end for
12:      $C_0$ = makespan ($S_0$)
13:      $S_{CI} = S_0$
14:      repeat
15:         neighborhood selection:
16:            $height \in (1 + (1 - T).(height_{max} - 1), height_{max})$
17:         generate new solution:
18:            for $j$ = 1 to 3 do
19:                if $Random(0, 1) < T$ then
20:                     mutate $S_{CI}$ according to mutation type $j$
21:                end if
22:            end for
23:            $C_{CI}$ = makespan ($S_{CI}$)
24:                if $C_{CI} \le C_{CI} - 1$ then
25:                     accept $S_{CI}$ as the new solution
26:                else if $\exp(-(C_{CI} - C_{CI-1})/T) > Random(0, 1)$ then
27:                     accept $S_{CI}$ as the new solution
28:                end if
29:            $CI = CI + 1$
30:            update $T$
31:      until stop criterion
32:   end procedure
```

3 Experimental results

To evaluate the performance of the proposed method, GOSA is applied on various benchmarks and synthetic graphs considering several target systems types and architectures. The real-life task graph benchmarks include LU decomposition and Gauss–Jordan Elimination (GJE) [37] with various task numbers as two well-known standard test problems. Moreover, we produced randomly generated task graphs with 20–100 task nodes as synthetic test cases. We compared the GOSA with three successful methods in terms of both the quality andexploration time of them and also with three recent algorithms in terms of performance. In the following subsection, an experimental design is presented to state the specifications and objectives of the experiments clearly.

3.1 Experimental design

To evaluate the proposed method, a total of ten different experiments were performed whose design specifications and objectives are presented in Table 1. It should be noted that due to the randomness of the algorithms' process, and the elimination of the role of chance and probabilities in the results and their interpretations, all experiments were performed with multiple repetitions and on different scenarios in terms of the graph and system size, which is clearly stated for each experiment." Moreover, in each experiment, this issue was emphasized, and the number of repetitions and consideration of different conditions of the graph or system were highlighted.

In the following, each of these tests is presented along with the details and results obtained and their interpretations.

Table 1 Specifications and objectives of the experiments

Exp. number	The experiment objective and the evaluation criteria	Target system configuration	Task graph types and characteristics	Number of repetitions	The algorithms
1	Tuning the value of the parameter k in the cooling function	$NT = 4, NC = 2$	Four random DAGS with 20–80 nodes	30	GOSA
2	Makespan, scheduling error	[NT, NC] in the range: [1, 4] and [1, 5]	20, 40, and 60 nodes graphs, each including ten types in terms of execution time and the required configuration nodes	10	Exhaustive Search, List-based heuristic, GA, and GOSA
3	The run-time and makespan improvement over the List Scheduler	$NT = 4, NC = 2$	Five various random graphs with 20–60 nodes	20	Exhaustive Search, GA, and GOSA
4	Convergence rate, average, worst, and best makespan	$NT = 4, NC = 2$	Five various random graphs with 20–60 nodes	50	The best, the worst, and average solutions of GA and GOSA
5	Relative error and the stopping iteration number	$NT = 4, NC = 2$	Five various random graphs with 20–60 nodes	5	Exhaustive Search, List-based heuristic, GA, and GOSA
6	Time scalability	$NT = 4, NC = 2$	Five various random graphs with 20–60 nodes	20	GA and GOSA
7	Confidence level, average, maximum, and minimum of the makespan and execution time	[NT, NC] in the range: [3, 6] and [1, 3]	25 random task graphs with 20–60 nodes	100	Exhaustive search, GA, and GOSA
8	Reliability of the results	$NT = 3, NC = 1$	20-node random DAGs	1000	List-based heuristic, GA, and GOSA
9	Makespan and execution time comparison to state-of-the-art and successful algorithms in the literature	$NT = 4, NC = 2$	GJE and LU decomposition graphs	30	DRNN-BWO, DCHG-TS, GAACO, FATS, BGA, HPSO-GA, and GOSA
10	Investigate the impact of various system sizes on the performance	NT in the range: [4, 12] and NC in the range [2, 6]	GJE graph with 645 and 820 nodes, and LU decomposition graph with 464 and 819 tasks	5	FATS, BGA, HPSO-GA, and GOSA

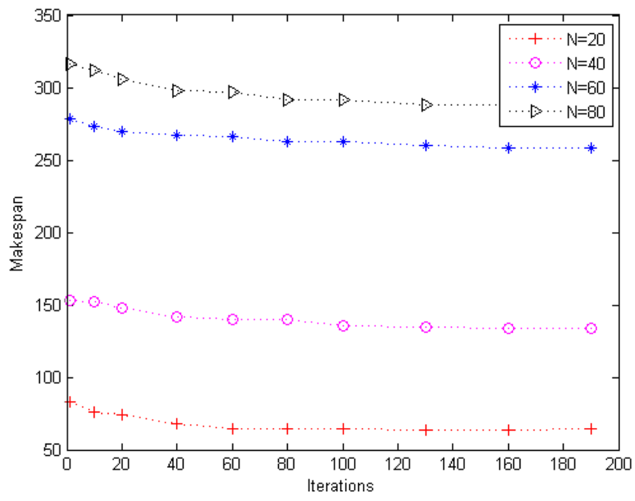


Fig. 6 Diagrams of makespan versus iterations for four different graphs

3.2 Results and discussion

3.2.1 Experiment 1

In the first experiment, the objective was to determine the value of the parameter, k, in the cooling function. To obtain a suitable value for this parameter, a reconfigurable system having four tiles and two controllers is supposed, and the diagram of makespan versus GOSA iterations for four graphs with a different number of tasks is shown in Fig. 6. As shown in the figure, by increasing the number of iterations, the makespan of the scheduling solutions decreases. To minimize the role of chance, the simulation was repeated 30 times for each graph, and the results were averaged. This diagram shows that the best value for the k parameter is located on the knee of the curves. This fact implies that the value of k should be considered in the following range:

$$2 < k \leq 3 \tag{12}$$

Another point that should be noted about this graph is its slow convergence. The main reason is the lack of movement and action to improve the solutions, which returns to the essence and nature of the problem of scheduling dependent tasks on Dynamically Reconfigurable Hardware. Most of the makespan of the solutions to such a scheduling problem include the critical path tasks arranged sequentially on tiles. The length of the critical path determines and limits the optimal makespan. The scheduling algorithms try to accommodate other tasks (non-critical path tasks) in the empty spaces parallel to the critical path tasks, to make maximum use of the idle times of other tiles. Therefore, despite the existence of many permutations to move the arrangement of tasks of each potential response time, many of which are infeasible, and minimal modes can lead to improvement of the solutions, which requires a lot of searching and time to find a feasible and better solution. Hence, the convergence rate toward the optimal solution slows down.

3.2.2 Experiment 2

Since GA has been demonstrated as a successful algorithm for this problem in the literature and provides high-quality solutions [20, 34]. In this set of experiments, the results of the proposed method, GOSA, were compared with those of a standard GA with optimized parameters' values. The List-based heuristic is another comparison method because it produces feasible scheduling solutions in a very short time. In fact, in comparison with other well-known scheduling algorithms, it has much lower-time complexity. To obtain the results of the three mentioned algorithms, optimal scheduling results are also needed, which are generated using Exhaustive Search. The Exhaustive Search was adopted as a criterion to calculate the scheduling error of the three other algorithms. The scheduling error of a scheduling algorithm concerning the optimal solutions generated by an Exhaustive Search can be computed using the following relation:

$$Scheduling\ Error = \frac{Algorithm' sMs - ES' sMs}{ES' sMs} \times 100 \tag{13}$$

where ES and Ms stand for Exhaustive Search and makespan, respectively. In this set of experiments, the proposed method is evaluated by testing on various random task graphs and compared by GA, List Scheduler, and the Exhaustive Search results. Since GA parameters, i.e., crossover and mutation rates, considerably affect the results [26, 31, 32], various simulation scenarios were examined to tune the GA parameters. This procedure is lengthy, and hence, only the final obtained optimal values are presented for the algorithm's parameters as below:

- Population size: 32.
- The number of generations: 100.
- Selection rate: 0.30.
- Mutation probability: 0.10.

These values are obtained after numerous runs of the GA with different parameter values, and the optimum ones were used in the experiments. By this, we are sure that the results returned by GA are its bests.

The number of tiles of each task and their execution time are randomly generated with uniform distribution in the ranges of [1, 4] and [1, 5], respectively. In all experiments, if stated else, the number of controllers, NC, was considered 2, and the number of tiles, NT, was taken 4.

To compare the methods in terms of makespan, the proposed method is evaluated by exploiting three various DAGs, including 20, 40, and 60 task nodes, each of which contains ten different graphs in terms of execution time and the required configuration nodes (i.e., width and depth).

Table 2 The proposed method (GOSA), GA, List Scheduler, and Exhaustive Search makespan solutions

Graph number	Num. of tasks = 20				Num. of tasks = 40				Num. of tasks = 60			
	List	GA	GOSA	E.S	List	GA	GOSA	E.S	List	GA	GOSA	E.S
1	75	69	67	63	194	176	176	157	264	249	246	226
2	70	63	59	55	182	168	161	148	291	271	271	243
3	84	79	76	74	168	155	154	136	254	241	237	221
4	84	77	72	68	175	164	161	148	278	263	261	243
5	74	68	63	60	170	155	154	140	272	256	254	230
6	86	80	76	75	182	166	165	152	259	241	240	220
7	83	76	73	70	191	174	174	158	289	271	274	250
8	69	66	62	60	172	161	161	146	295	275	272	249
9	92	87	83	79	183	172	171	162	269	251	250	233
10	94	83	80	71	187	173	172	157	282	266	270	247

Table 3 The run-time and makespan improvements over the List Scheduler for 30 iteration numbers

Graph number	Number of tasks	E.S.		GA		GOSA	
		Time (min)	Improvements (%)	Time (min)	Improvements (%)	Time (s)	Improvements (%)
1	20	7.3	24	0.8	10	1.3	10
	40	17.2	13	2.4	4	4.3	5
	60	26.8	16	3.9	6	7.4	7
2	20	9.3	12	1.3	4	2.5	4
	40	17.6	16	2.2	5	4.1	6
	60	27.4	12	3.9	4	7.2	4
3	20	7.6	23	1	9	1.8	10
	40	16	19	2.1	5	3.9	7
	60	25.9	18	3.7	7	6.8	9
4	20	8.2	14	1	5	1.7	12
	40	17.4	22	2.5	9	4.7	9
	60	26.2	20	3.7	7	6.6	5
5	20	7.9	19	0.8	6	1.2	9
	40	16.4	20	2.2	7	4.1	10
	60	26.7	22	4	6	7.6	7

Table 4 The run-time and makespan improvements over the List Scheduler for 100 iteration numbers

Graph number	Number of tasks	ES		GA		GOSA	
		Time (min)	Improvements (%)	Time (min)	Improvements (%)	Time (s)	Improvements (%)
1	20	22.2	31	2.5	13	4.6	12
	40	56.4	22	7.1	7	13.5	8
	60	81.5	20	11.8	8	23	7
2	20	22.8	24	2.6	9	4.8	12
	40	52.3	18	6.8	7	12.4	9
	60	86.5	19	12.3	7	25.2	7
3	20	22.5	29	2.4	15	4.5	17
	40	52.9	14	6.7	5	12.1	5
	60	85.1	20	12.1	8	24	9
4	20	27.1	18	2.8	7	5.1	9
	40	52.2	16	6.6	6	11.6	9
	60	92.6	18	12.6	6	27	5
5	20	25.3	23	2.7	10	4.9	14
	40	55.5	19	7	8	13.7	9
	60	92.6	13	12.7	4	27.3	5

The experiment is repeated ten times, and the results obtained by each method were averaged. The makespan provided by each method is reported in Table 2. The results show that the List Scheduler, GOSA, and GA average error over the Exhausting Search is 12.3%, 7%, and 8%, respectively.

3.2.3 Experiment 3

In the next set of experiments, we ran the algorithms for a constant number of iterations and computed the results of each method to find out which one performs better. These experiments were done for different iteration numbers on three sets of multiple graphs with 20–60 tasks, including

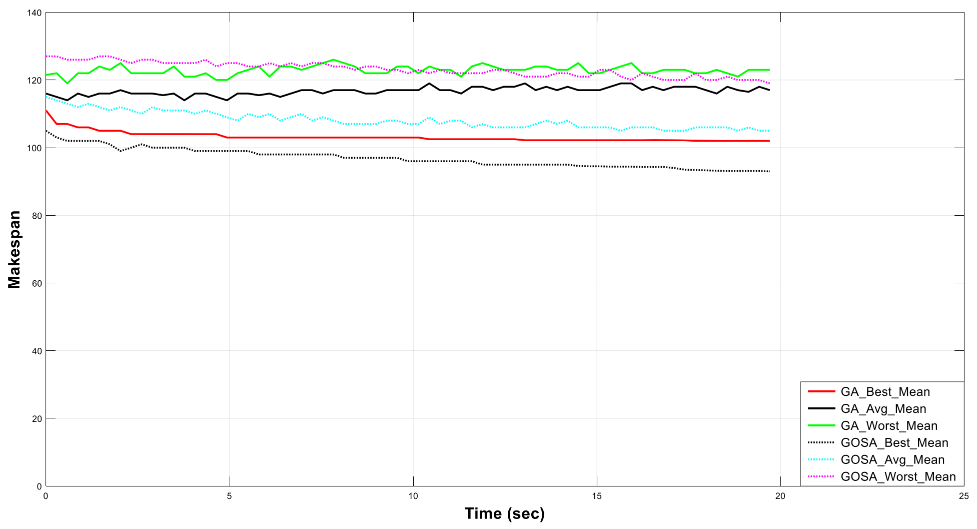


Fig. 7 Population mean plots of average, worst, and best makespan produced by GOSA and GA over time

five various graphs. The experiments for 30 and 100 iteration numbers were repeated 20 times for each algorithm, and their run-time, besides the averaged makespan results in the form of improvements compared to those of List Scheduler, are presented in Tables 3 and 4.

As can be inferred from these results, the quality of the solutions provided by GA, GOSA, and ES is average 7.2%, 8.1%, and 19.1% higher than that of the List Scheduler, respectively. Moreover, GOSA is 31 times faster than GA on average in terms of the algorithm execution time. The reason behind the fact that the quality of the solutions generated by GA is lower than those of GOSA is that the GA is not tuned and optimized for the problem. This means that it might include all unrelated cases so that all tasks would be arranged consecutively on a single tile in extreme cases. In return, our proposed algorithm tries to fill unoccupied tiles to produce a higher-quality population in the early stages of the algorithm and thus results in a quicker convergence speed.

3.2.4 Experiment 4

To verify the results of the previous experiment, another test was carried out on five various graphs five times, and all algorithms were allowed to resume for 20 s. The results of this experiment are shown as the population mean plots of the average, worst, and best makespan obtained by GOSA and GA over time in Fig. 7.

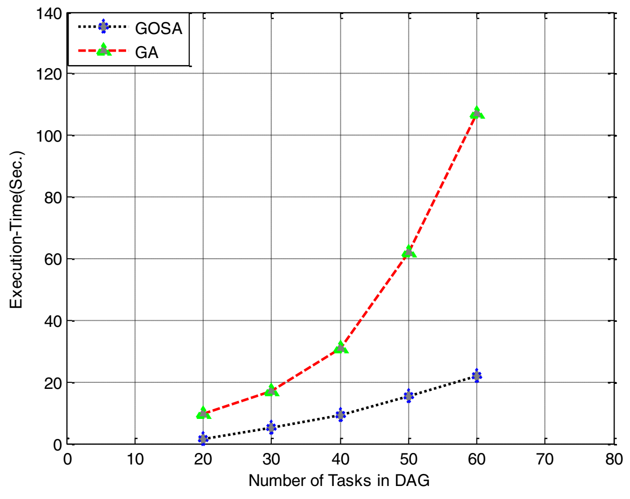


Fig. 8 Time scalability of the algorithms

As can be seen from the figure, the quality of the GOSA-Avg-Mean is improved, unlike those of GA, which became worse over time. There are some reasons behind this

Table 5 The algorithms' relative error and their stopping iteration number for five different DAGs

Task set	Number of tasks	*GOSAerror*%	GOSA stop iteration num	*GAerror*%	GA stop iteration num	*Listerror*%
1	20	3.57	96	5.26	41	15.14
2	30	7.32	125	6.75	47	15.84
3	40	9.91	116	9.14	59	16.02
4	50	10.17	139	11.69	67	18.17
5	60	13.12%	147	13.10	72	18.79

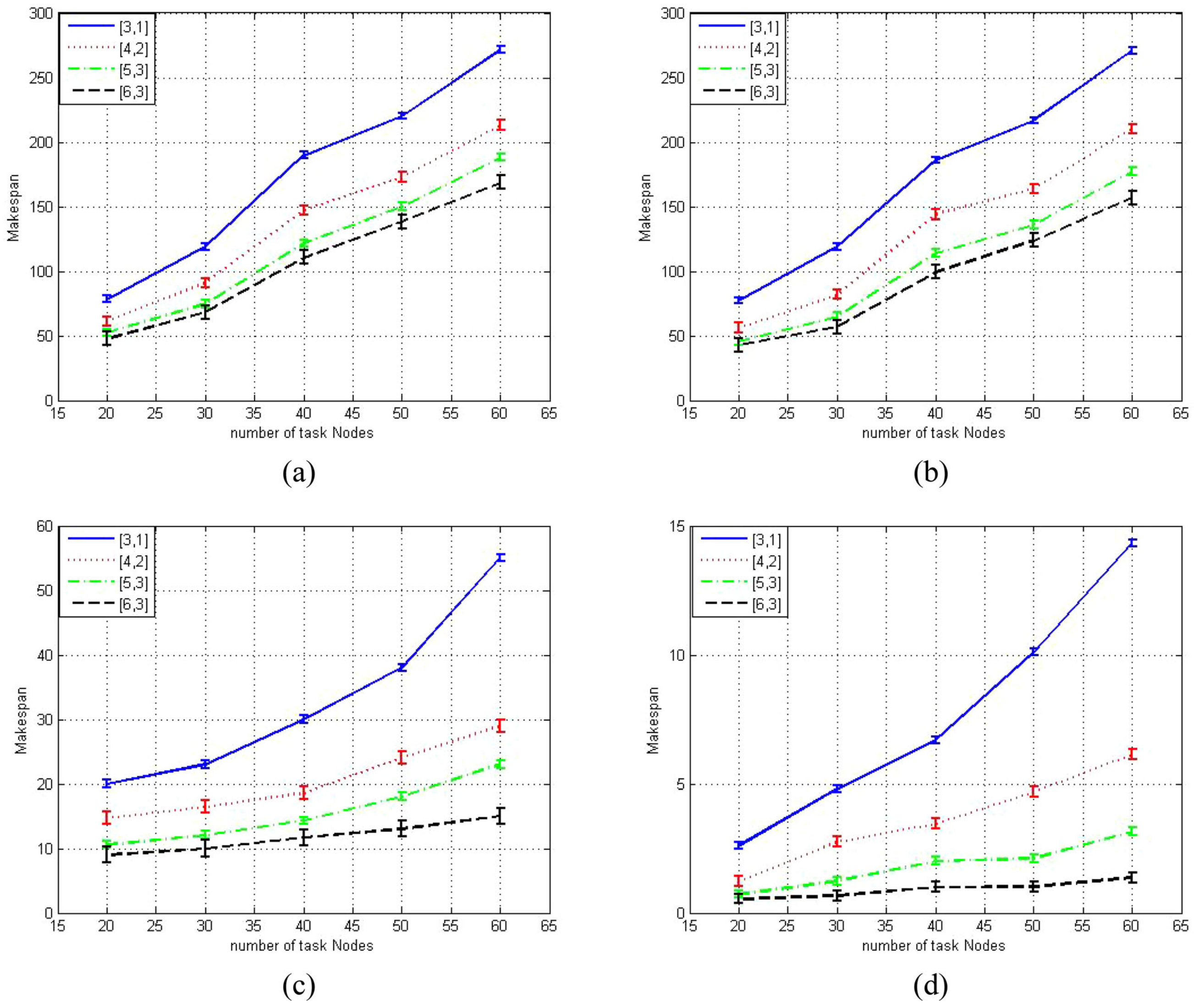


Fig. 9 The confidence level of the solutions generated by **a** GA in terms of the quality, **b** GOSA in terms of the quality, **c** GA in terms of the execution time, and **d** GOSA in terms of the execution time, for different scenarios

observation. First, the dominant effect of the mutation operator for evolving the algorithm in every generation. Second, is that the crossover operator, due to its need for the repair phase, has little impact on enhancing and converging the population. This issue can be relaxed and tackled by adapting the GA for scheduling problems by exploiting some techniques, such as Genotype-to-Phenotype Mapping (GPM) [17]. In addition to the single-solution nature of GOSA, another reason that GOSA is much faster than population-based GA is the necessity of repairing steps to repair violated precedence constraints resulting from gene ordering disturbances during crossover and mutation operations. However, GOSA, by exploiting the S-graph and the nodes' height value, always preservers the precedence constraints and produces valid solutions. Therefore, there is no need for any extra repairing phases. Moreover, the non-optimality of direct use of GA for the problem, which resulted in the presence of many inconsistent solutions in the population, is another reason for the deterioration of the quality of the solutions over time.

3.2.5 Experiment 5

In the subsequent two experiments, we investigated the impact of increasing the size of the problem on both the algorithms' execution time and the quality of the solutions they generate. In the first experiment, the algorithms were run five times on five different graphs, and the stopping criterion was the fitness not changing over 30 successive generations. The target was to log the makespans of scheduling each DAG. The results of this experiment are presented in Table 5 as the algorithms' relative error compared to the solutions acquired by the ES.

The results show that the quality of the solutions provided by GOSA is comparable to or sometimes better than those of GA. As the graph sizes get larger, both algorithms require more iterations to be converged, but in an unscalable pattern. One may note that GOSA needs more iteration numbers to be converged in comparison with GA. Still, one should also note that more generation numbers do not necessarily mean greater execution time, and this parameter value must be calculated separately.

3.2.6 Experiment 6

Next is dedicated to measuring that value, and to do so, the algorithms were run to reach 95% of the optimum solutions found by ES for five different graphs. The average execution time of the algorithms by performing the test 20 times on each graph is presented as a diagram in Fig. 8.

The results show that GOSA can reach the specified quality for the solutions in a much shorter time in comparison with GA. For every ten additional nodes, it needs about 3.5 s more time which shows its relative linearity in the execution time. However, GA exhibits different behaviors in which its run-time is approximately linear, for graphs smaller than 40 nodes.

3.2.7 Experiment 7

In the next series of experiments which were done for five different system configurations, by considering *NT* ranging from 3 to 6 and *NC* varying from 1 to 3, we evaluated the confidence level of the simulations. In the first experiment, five different graphs with various numbers of tasks and five different system types are considered. The stopping condition was taken as fitness, not changing for 30 successive iterations. The tests are repeated 100 times on each DAG, and the average, maximum, and minimum of the algorithms' makespan are logged. In the second experiment, the previous scenario was repeated with the difference that the target was reaching 95% of the optimum values obtained by Exhaustive Search. Finally, the results of both experiments are presented as diagrams in Fig. 9.

As inferred from the diagrams, the gap between the best and the worst solutions for every scenario is not significant in both experiments. Quantitatively, the average makespan and execution time error for GA are 3.35% and 3.89%, respectively, and those of GOSA are 2.79% and 1.33%, respectively. This verifies the validity of the results of the previous experiments and shows that both algorithms are implemented well, and the simulations have an acceptable confidence level.

3.2.8 Experiment 8

To clarify the effect of chance on the results of both GA and GOSA, we investigated the reliability of the algorithms. In this regard, we repeated both methods on a random DAG 1000 times, and the average, maximum, minimum, and standard deviation values of all results are recorded for their makespan. The results are shown as a candle diagram in Fig. 10.

3.2.9 Experiment 9

To evaluate the performance of the proposed algorithm, we have done some experiments on the graphs of real benchmark problems, i.e., GJE and LU decomposition according to Table 6. Then, the results are compared with those of some state-of-the-art algorithms in the field. The selected and compared algorithms are FATS [22], HPSO-GA [23], and BGA [24], which are reviewed in the Related Work section. To have a just comparison, all the competitive algorithms are used based on their optimized parameter

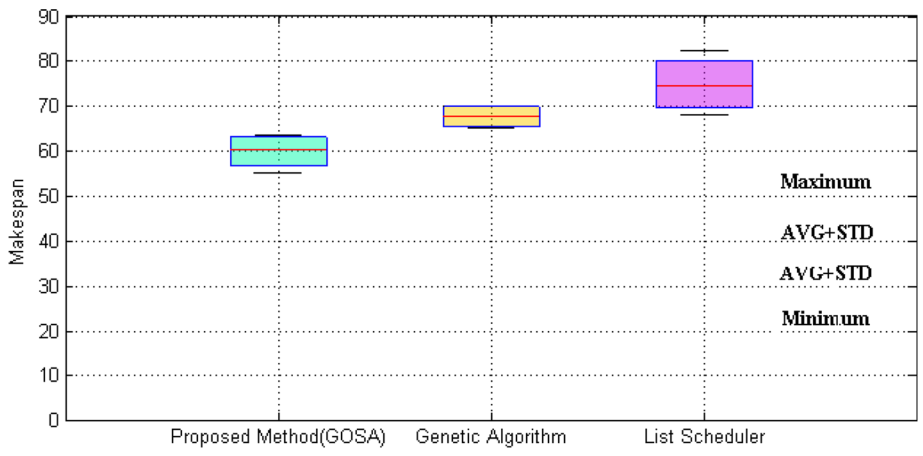


Fig. 10 Comparison of the reliability of GOSA, GA, and the List Scheduler by applying them on a random DAG 1000 times

Table 6 Parameters' values of the standard task graphs in the experiments

Problem	Number of tasks	Computation time	Communication time
GJE	15, 21, 28, and 36	40 s/task	100 s
LU decomposition	14, 20, 27, and 35	20 s for bottom layer tasks, plus 10 s for all other layers' tasks	80 s

Table 7 Optimized parameters' values for all compared algorithms

FATS	BGA	HPSO-GA
Num. of ants: 16	*GNS* and *GNP*: 40	C_1: 2
α: 0.02	*SGNS* and *SGNP*: 10	C_2: 2
Q: 3	*PsS* and *PsP*: 20	ω: 0.9
	PmS: 0.32, *PmP*: 0.2	Crossover rate: 0.7
	PcS: 0.74, *PcP*: 0.8	Mutation rate: 0.15

values according to Table 7, which are obtained from their references, and the maximum iteration number is considered as 100.

The values of the parameters used for these standard graphs, such as the number of tasks, computation, and communication time, are presented in Table 7.

In the first experiment, our algorithm, besides the other mentioned three algorithms, was run to schedule the benchmark graphs on an RC system with $NC = 2$ and $NT = 4$. The results of this experiment which are the average of running each algorithm 30 times are presented in Fig. 11.

The results show that GOSA and FATS are superior to other algorithms in all cases.

However, the former performs better for larger problem sizes. Numerically, for different GJE graph sizes, ours is, on average, 5%, 13%, and 17% outperformed FATS, BGA, and HPSO-GA, respectively. For the different LU decomposition graph sizes, GOSA surpasses FATS, BGA, and HPSO-GA, by 3%, 12%, and 16%, respectively. As can be inferred from the results, FATS also generated high-quality solutions due to its customized graph translation mechanism, which shrinks the search space, including only the promising parts of the space. There is an important observation in the results; the quality of the solutions provided by GOSA and FATS gets better by increasing the size of the problems. This is because by increasing the graph sizes, the search space would be expanded. Still, these algorithms remove the infeasible defective solutions from the solution space and only search the most promising zones. On the contrary, the other algorithms, due to the lack of such mechanisms and also due to their consecutive population-based algorithms, require more time to be converged. Therefore, they fail to compete with ours in a fixed number of generations.

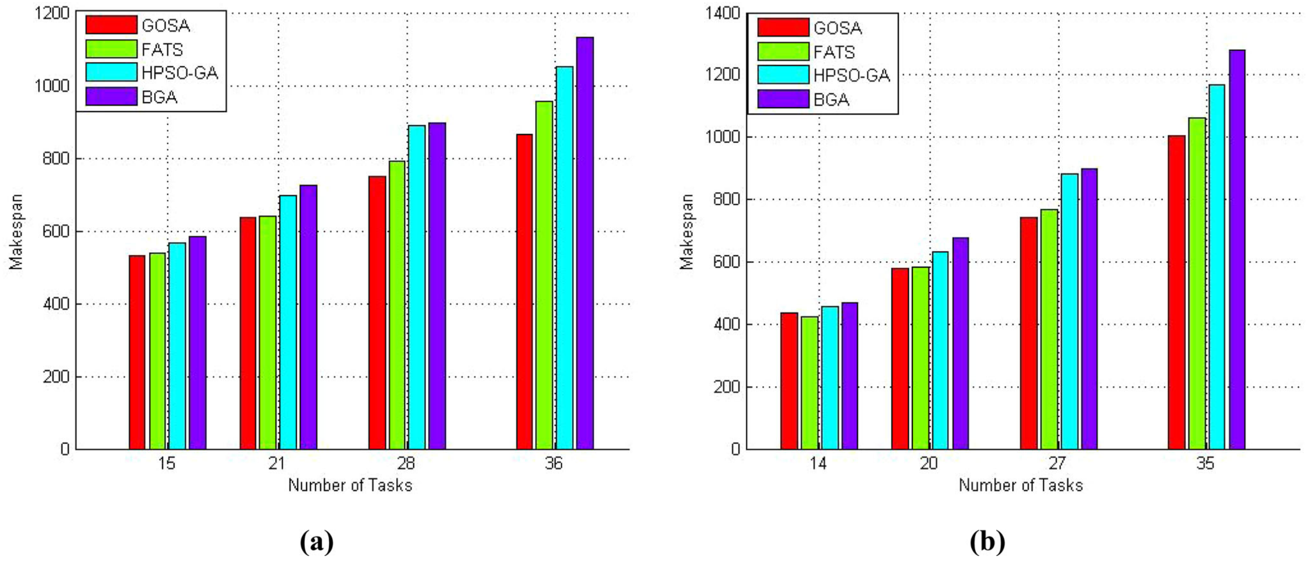


Fig. 11 Makespan of the algorithms for different sizes of **a** GJE graph and **b** LU decomposition graph

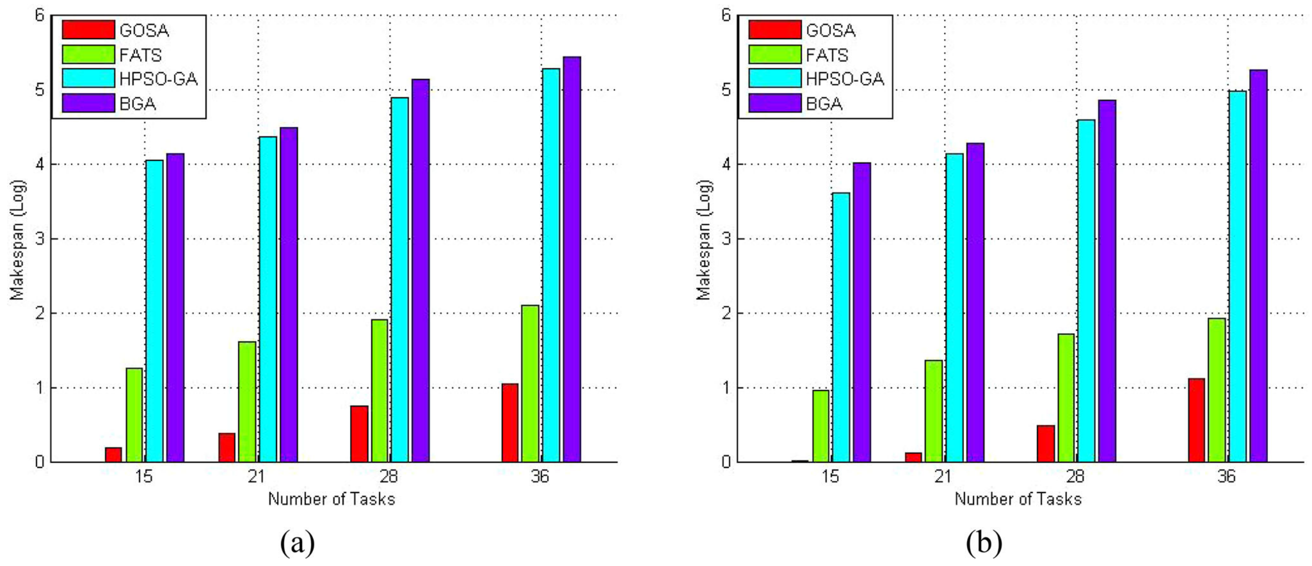


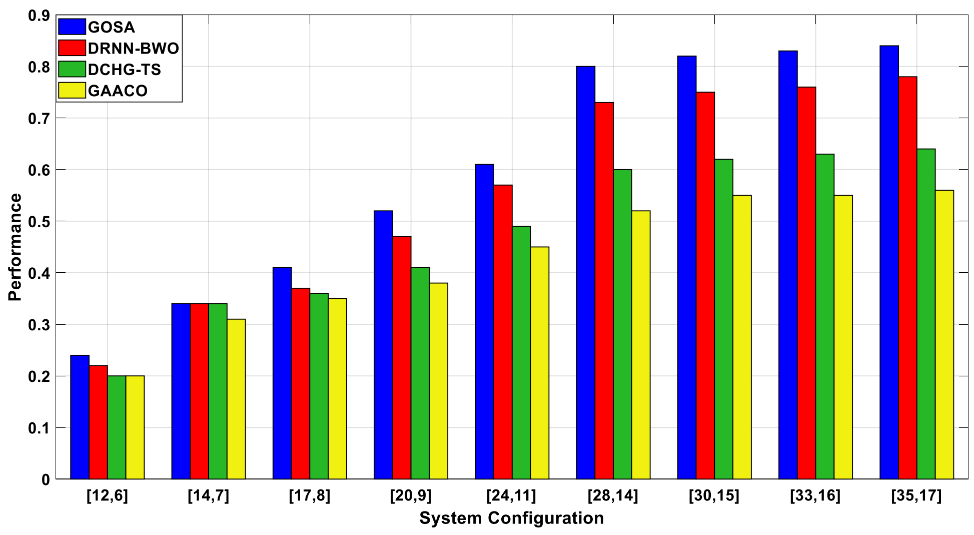
Fig. 12 Algorithms' run-time to find final scheduling for different **a** GJE graph sizes and **b** LU decomposition graphs

To clarify this point, another experiment is done in which the execution time of the algorithms for finding an assumed makespan is measured and reported in the graph of Fig. 12.

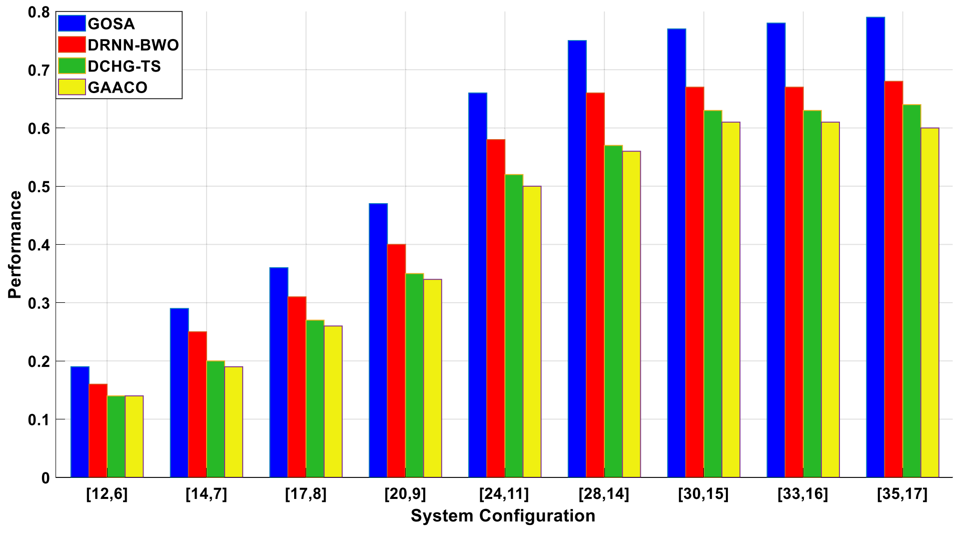
The results show that the run-time of GOSA is significantly lower than the others. The reason is that the proposed algorithm explores the search space according to its graph specifications which guarantee the feasibility of the solutions while speeds-up the search process. This results in a high convergence rate and also a low-time complexity. Among the other algorithms, FATS also has a considerably low execution time which is the result of its TSP-likened construction graph generation mechanism. It allows those solutions to be generated that preserve the system limitations and DAG precedence constraints. The other two methods are very time-consuming due to their successive and independent population-based algorithms. BGA uses two consecutive GAs, with parameters to be adjusted, and includes too many individuals in each generation which require cost evaluation calculations. Moreover, it exploits particular types of operators to evolve the generation members while requiring extra repair phases and procedures to correct the infeasible chromosomes. Similarly, HPSO-GA also faces challenges and drawbacks, leading to high-time complexity.

The most important reason for the high quality of the solutions provided by GOSA is its ability to access the entire search space, as opposed to other methods that limit the search space or remove some of its zones. Those population-based algorithms divide the problem into subproblems or split the search process into distinct consecutive phases to reduce the volume of calculations and the execution time of the algorithms. By this, the tasks of exploration and exploitation of the search algorithm are separated from each other. This prevents the algorithms from accessing some promising parts of the search space and consequently degrades the quality of the generated solutions. However, in GOSA, the search space is not restricted or excluded, and the whole search space is available for the algorithm. But still, two main challenges should be tackled to achieve such competency. First, the SA is inherently a single-solution algorithm, and its steps from one iteration into another are very short. Second, a considerable part of the search space lies in the infeasible region due to the presence of solutions that violate the precedence constraints, and entering those search areas wastes the algorithm's execution time. To tackle the mentioned challenges, two innovative mechanisms are presented. First, the step lengths of the algorithm are adjusted by proposing a customized neighborhood selection mechanism and a problem-dependent cooling function. By the progress of the algorithm, these functions control neighborhood zone selection and also the algorithm's step lengths based on the introduced parameters. Second, an innovative graph and solution structure called schedule graph (S-graph), which is fitted to the profiles of the reconfigurable system's task scheduling problem, is introduced. It guarantees the precedence constraints and the system limitations so that the algorithm will be practicable for various target systems. Moreover, it is compatible with applications with different graph structures.

Another set of experiments is performed in which the performance of the proposed algorithm is compared with those of some state-of-the-art algorithms. The evolutionary-based competitor algorithms are DRNN-BWO [38], DCHG-TS [39], and GAACO [40]. This set of experiments



(a)



(b)

Fig. 13 Performance comparison of the algorithms for different sizes of **a** GJE task graph with 465-task nodes and **b** LU decomposition task graph with 820 nodes

is designed so that the effect of increasing the system size, i.e., increasing the number of *NT* and *NC*, on the efficiency of the algorithms is investigated. *NT* was ranging from 45 to 90 and *NC* was varying from 12 to 40. The system configuration is considered as [*NT*, *NC*] as [6–9, 11, 12, 14, 14–17, 17, 20, 24, 28, 30, 33, 35] in each experiment. The performance parameter measured in these experiments is makespan, a 465-task node GJE graph and an 820-task node LU decomposition graph are considered target applications, and the number of tiles and controllers are adjustable parameters. The results of these experiments are shown in Fig. 13.

As shown in Fig. 13, the algorithms' efficiency improves with an increasing number of tiles and controllers due to increased processing parallelism. However, by increasing the number of tiles and controllers, efficiency improvement is limited to a certain number of processors. This is due to the serial nature of the task graphs, as well as the amount of overhead imposed on the system. The main reason for the superiority of our proposed algorithm is the introduction of innovative graph and solution structures, S-graphs. As said before in the previous section, such a structure is fitted to the profiles of the reconfigurable system's task scheduling problem and guarantees the precedence constraints and system limitations. Therefore, the algorithm will be practicable for various target systems with different types and features as well as for various applications with different graph structures. However, other algorithms' potential solutions are entirely generated randomly, and there are no customized mechanisms for

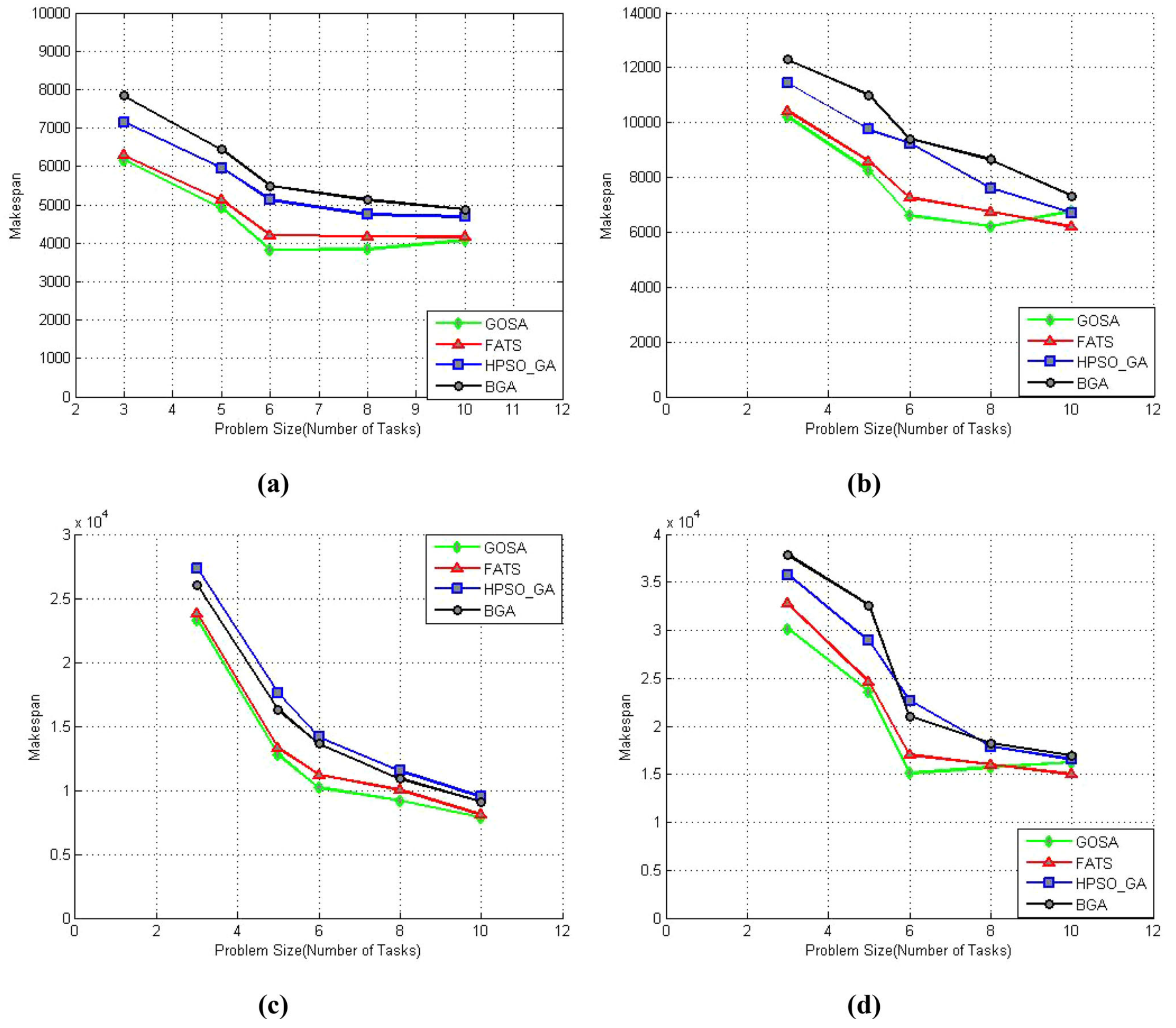


Fig. 14 Scheduling makespan for **a** GJE task graph with 465-task nodes, **b** GJE task graph with 820-task nodes, **c** LU decomposition task graph with 464-task nodes, and **d** LU decomposition task graph with 819-task nodes on different target system sizes

distributing tasks among tiles while considering their configuration prefetching.

In Fig. 13a and b, when the system size is minimum, i.e. [6, 12], in terms of average performance, our proposed algorithm outperformed DRNN-BWO, DCHG-TS, and GAACO by almost 15%, 34%, and 35%, respectively. However, as the number of tiles and controllers increases, the efficiency rate increases dramatically. When the number of processors equals [14, 27], experimental results show that the proposed method effectively improves average performance by at least 14%, 31%, and 39% compared to DRNN-BWO, DCHG-TS, and GAACO, respectively. In the case of the maximum system size, i.e. [16, 34], GOSA improves the performance by 11%, 28%, and 39%, compared with DRNN-BWO, DCHG-TS, and GAACO, respectively.

3.2.10 Experiment 10

To investigate the impact of various system sizes on the performance of the proposed method, we have done another experiment using very large GJE and LU decomposition graphs. In other words, we tried to evaluate the scalability of the algorithms by measuring the makespan reduction by increasing the target system size. The makespan results on target systems with 2–6 controllers and 4–12 tiles for GJE task graphs of 645 and 820 tasks, and the LU decomposition task graphs of 464 and 819 tasks are shown in Fig. 14, respectively.

The descending pattern of all diagrams was already apparent. Still, the critical point is the greater slope of the GOSAs makespan results according to the growth of the system size compared with those of others. It means that our proposed method offers cheaper solutions to dealing with large problems. Another observation is the reasonably uniform reduction in all algorithms' makespan, but the emergence of saturation in the proposed method's results in the system with four tiles and two controllers is notable. Furthermore, for extra-large problems, GOSA generates better solutions in comparison with others, while exhibiting a linear behavior as the target system size increases. The improvement of the quality of the solutions provided by GOSA compared to the FATS method was limited to 6% on average. However, the noteworthy point is that the proposed method, despite being a single-solution approach, thanks to its innovative graph-oriented solution structure could reach such quality against a successful population-based approach. The quality improvement rate compared to the other two methods, i.e., BGA and HPSO_GA, is very promising due to the confined search space that is resulting from the nature of these two-stage algorithms.

As can be seen from these candle diagrams, the stability of both GA and GOSA is acceptable so that the differences between their best and the worst results are narrow. However, since the proposed method uses its special graph-oriented searching and pruning mechanism, its reliability is somehow better.

4 Discussion

As can be inferred from the results, the proposed method provides some useful advantages. The most important strengths of the proposed method are its low-time complexity and its fast convergence rate despite being based on a single-solution search method. The other strength is its dynamic change of the search steps in terms of neighborhood size in each iteration based on the problem structure. By the problem structure, we mean the topology and the number of graph nodes and also the size of the reconfigurable hardware system. Moreover, it ensures that infeasible answers are not produced in each iteration by using an innovative mechanism including S-graph, T-strings, and C-strings which preserve the precedence constraints in both DAG tasks and configurations. Therefore, it makes us unnecessary to repair the invalid potential solutions in each iteration. Although the proposed method has many advantages and strengths, in some special cases, its efficiency decreases. Since one of the control operators of the proposed method in each iteration is based on the node height; if the structure and topology of a given DAG are such that the number of nodes of one level of the graph is very large and also the nodes of that level are the same in terms of size, the possibility of changing the arrangement of a large percentage of them in the scheduling string in the final iterations is reduced. Moreover, we emphasize that the proposed method is designed for reconfigurable systems with identical homogeneous tiles. If the system is heterogeneous, it is necessary to make modifications to its structure and algorithm.

Moreover, the analytical comparison of the computational complexity of the proposed method with other methods can be useful too. To this end, we examined the time complexity of the algorithms in the following. In the FATS algorithm [24], assuming that n is the number of a finite set of interconnected tasks, m is the number of ants, t is the number of available tiles for executing each node, and I_{max} is the maximum iterations, the overall time complexity of the algorithm is $O(mn^2tI_{max})$. In the other method, the BGA [24], assuming that the population sizes of the first and second phases are p and p', respectively, the number of task graph tasks is denoted by n, which are connected by e edges, and g_{max} is the maximum number of generations, the number of processors is considered as t, and the overall time complexity of the algorithm is $O(g_{max}pp'n^3t)$. The other competitor algorithm, HPSO-GA [23], is very similar to the previous one in its second phase but uses a PSO algorithm in its first phase. PSO has the time complexity of $O(pnt)$, where p denotes the number of particles, n is the number of tasks, and t is the number of processors. However, in the second phase, GA imposes the time complexity in the order of $O(g_{max}pn^2t)$. In the proposed method, which is based on a single-solution SA algorithm, the first step aims to initialize the solution of the first iteration, S_0. This step has a time complexity of $O(n+t)$, assuming that n is the number of tasks in the task graph, and t is the number of processors or tiles. Then, the cost of the generated solution will be evaluated by order of $O(1)$. Developing the S-graph in the next step requires investigating the dependency links between nodes of the graph, which has a time complexity of $O(e)$. Now, the algorithm should evolve the current solution by mutating some of its elements based on the height value parameter. Such a parameter is a priority queue that contains all ready tasks in the neighborhood of the nodes at any given instant. To implement this queue, a binary stack is used, which has a time complexity of $O(\log n)$. Computing the cooling plan as a temperature parameter has $O(1)$ time complexity. By repeating the algorithm for the I_{max} number of iterations, the overall time complexity of the algorithm can be defined as $O(I_{max}n\log n)$. Therefore, while the time complexities of BGA, HPSO-GA, and FATS are proportional to n^3, n^2, and n^2, respectively. The time complexity of the proposed algorithm is proportional to $n \log n$, which verifies its speed and lower execution time in comparison with others.

5 Conclusions

In this paper, an adapted and customized Simulated Annealing (SA) algorithm is presented to efficiently solve the Dynamically Reconfigurable Hardware (DRHW) task scheduling problem. The search space of this problem is multi-dimensional and extremely large due to the existence of configurations besides tasks that need to be scheduled. At the same time, their precedence constraints must be satisfied, and the overall makespan should be minimized. To consider both tasks and configurations simultaneously, a new graph called an extended graph is first generated. Then, two-dimensional strings are coded as potential solution schemes. In the proposed algorithm, GOSA, to obtain the optimal solution with the least execution time, a graph-oriented approach is developed, which selects a dynamic neighborhood for picking the next node according to the graph topology and task sizes. By using an adaptive problem-dependent cooling function, the neighborhood range is gradually shrunk as the algorithm progress. Then, the node selection mechanism in each iteration is done by exploiting three innovative inventive genetic operators. This neighborhood selection mechanism and the multi-fold operators cover the vast and complex problem space and increase the proposed algorithm's performance. Simulation experiments on different benchmarks and randomly generated datasets with different sizes proved the efficiency and scalability of GOSA. Comparison results with the basic and also state-of-the-art algorithms indicate that GOSA outperforms other algorithms in terms of execution time. At the same time, its scheduling quality is better in most cases, and in some others, it is comparable to the best competitor. Quantitatively, the average improvements of GOSA, GA, and the Exhaustive Search over the List Scheduler for task graphs with 20–60 nodes are 7.2%, 8.1%, and 19.1%, respectively. Moreover, the proposed method is on average 31 times faster than GA. In other sets of experiments, we have done numerous tests on real benchmark task graphs and compared the results with those of some state-of-the-art algorithms. The results of these experiments show that the solutions provided by our proposed method outperform FATS, HPSO-GA, and BGA on average by 5%, 13%, and 17% in terms of the quality of the solutions. Also, its run-time is extremely lower than those. Moreover, the performance of the GOSA is on average, 11%, 28%, and 36% better than those of DRNN-BWO, DCHG, and GAACO, respectively. In the future, we intend to improve the method to be able to perform run-time scheduling. We also plan to consider other parameters involved in reducing bottlenecks and increasing the overall efficiency of the system, including task data locality and running task preemption.

Funding The authors did not receive support from any organization for the submitted work.

Data availability The data that support the findings of this study are available from the authors upon reasonable request.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Rabozzi M (2020) Caos: cad as an adaptive open-platform service for high performance reconfigurable systems. In: Special topics in information technology. Springer, Cham, pp 103–115
2. Chen S, Huang J, Xu X, Ding B, Xu Q (2018) Integrated optimization of partitioning, scheduling, and floorplanning for partially dynamically reconfigurable systems. IEEE Trans Comput Aided Des Integr Circuits Syst 39(1):199–212
3. Saha S, Zhai X, Ehsan S, Majeed S, McDonald-Maier K (2021) RASA: reliability-aware scheduling approach for FPGA-based resilient embedded systems in extreme environments. In: IEEE transactions on systems, man, and cybernetics: systems
4. Chen Z, Qiu M, Ming Z, Yang LT, Zhu Y (2013) Clustering scheduling for hardware tasks in reconfigurable computing systems. J Syst Architect 59(10):1424–1432
5. da Silva EC, Gabriel PH (2020) A comprehensive review of evolutionary algorithms for multiprocessor DAG scheduling. Computation 8(2):26
6. Srikanth GU, Geetha R (2018) Task scheduling using Ant Colony Optimization in multicore architectures: a survey. Soft Comput 22(15):5179–5196
7. Nayak S, Panda M (2020) Hardware partitioning using parallel genetic algorithm to improve the performance of multi-core CPU. In: Advances in intelligent computing and communication. Springer, Singapore, pp 467–474
8. Hou N, He F, Zhou Y, Chen Y (2020) An efficient GPU-based parallel tabu search algorithm for hardware/software co-design. Front Comp Sci 14(5):1–18
9. Pal H, Rohilla B, Singh T (2018) A Deadline-aware modified genetic algorithm for scheduling jobs with burst time and priorities. In: Nature inspired computing. Springer, Singapore, pp 55–67
10. Bandaru S, Deb K (2016) Metaheuristic techniques. In: Decision sciences. CRC Press, pp 709–766
11. Wolpert DH, Macready WG (1997) No free lunch theorems for optimization. IEEE Trans Evol Comput 1(1):67–82
12. Mollajafari M, Shahhoseini HS (2016) An efficient ACO-based algorithm for scheduling tasks onto dynamically reconfigurable hardware using TSP-likened construction graph. Appl Intell 45:695–712
13. Boussaïd I, Lepagnot J, Patrick S (2013) A survey on optimization meta-heuristics. Inf Sci 237:82–117
14. Li SG, Feng FJ, Hu HJ, Wang C, Qi D (2014) Hardware/software partitioning algorithm based on genetic algorithm. J Comput 9(6):1309–1315

15. Xian TR, Halim ZA, Leong CC, Gim TJ (2021) Hardware-software partitioning using three-level hybrid algorithm for system-on-chip platform. Bull Electr Eng Inf 10(1):466–473
16. Saha S, Sarkar A, Chakrabarti A, Ghosh R (2017) Co-scheduling persistent periodic and dynamic aperiodic real-time tasks on reconfigurable platforms. IEEE Trans Multi-Scale Comput Syst 4(1):41–54
17. Mollajafari M, Shahhoseini HS (2011) A repair-less genetic algorithm for scheduling tasks onto dynamically reconfigurable hardwar. Int Rev Comput Softw 6(2):206–212
18. Musa N, Gital AYU, Zambuk FU, Usman AM, Almutairi M, Chiroma H (2020) An enhanced hybrid genetic algorithm and particle swarm optimization based on small position values for tasks scheduling in cloud. In: 2020 2nd international conference on computer and information sciences (ICCIS). IEEE, pp 1–5
19. Zhang L, Zhou L, Salah A (2020) Efficient scientific workflow scheduling for deadline-constrained parallel tasks in cloud computing environments. Inf Sci 531:31–46
20. Huynh TTB, Pham DT, Tran BT, Le CT, Le MHP, Swami A, Bui TL (2020) A multifactorial optimization paradigm for linkage tree genetic algorithm. Inf Sci 540:325–344
21. Janakiraman N, Kumar PN (2014) Multi-objective module partitioning design for dynamic and partial reconfigurable system-on-chip using genetic algorithm. J Syst Archit 60(1):119–139
22. Mollajafari M, Shojaeefard MH (2021) TC3PoP: a time-cost compromised workflow scheduling heuristic customized for cloud environments. Clust Comput 24(3):2639–2656
23. Kumar N, Vidyarthi DP (2016) A novel hybrid PSO–GA meta-heuristic for scheduling of DAG with communication on multiprocessor systems. Eng Comput 32(1):35–47
24. Bonyadi MR, EbrahimiMoghaddam M (2009) A bipartite genetic algorithm for multi-processor task scheduling. Int J Parallel Prog 37(5):462–487
25. Shahhoseini HS, Saleh Kandzi E, Mollajafari M (2014) Nonflat surface level pyramid: a high connectivity multidimensional interconnection network. J Supercomput 67:31–46
26. Yu S, Li K, Xu Y (2018) A DAG task scheduling scheme on heterogeneous cluster systems using discrete IWO algorithm. J Comput Sci 26:307–317
27. Orsila H, Kangas T, Salminen E, Hamalainen TD (2009) Parameterizing simulated annealing for distributing task graphs on multiprocessor SoCs. In: Proceedings of the international conference on System-on-chip, pp 19–26
28. Mishra A, Mishra KS, Mishra PK (2019) Performance evaluation of simulated annealing-based task scheduling algorithms. In: International conference on information management and machine intelligence. Springer, Singapore, pp 145–152
29. Orsila H, Salminen E, Hämäläinen T (2013) Recommendations for using simulated annealing in task mapping. Des Autom Embed Syst 17(1):53–85
30. Houshmand M, Soleymanpour E, Salami H, Amerian M, Deldari H (2010) Efficient scheduling of task graphs to multiprocessors using a combination of modified simulated annealing and list based scheduling. In: Proceedings of the international symposium on intelligent information technology and security informatics, pp 350–354
31. Ferrandi F, Lanzi PL, Pilato C, Sciuto D, Tumeo A (2013) Ant colony optimization for mapping, scheduling and placing in reconfigurable systems. In: Proceedings of the adaptive hardware and systems (AHS), NASA/ESA Conference, pp 47–54
32. Ferrandi F, Pilato C, Sciuto D, Tumeo A (2010) Mapping and scheduling of parallel C applications with ant colony optimization onto heterogeneous reconfigurable MPSoCs. In: proceedings of the design automation conference (ASP-DAC), pp 799–804
33. Li Y, Wu M, Ye X, Li W, Xue R, Wang D, Fan D (2021) An efficient scheduling algorithm for dataflow architecture using loop-pipelining. Inf Sci 547:1136–1153
34. Qu Y, Soininen JP, Nurmi J (2006) A parallel configuration model for reducing the run-time reconfiguration overhead. In: Proceedings of the conference on design, automation and test, pp 965–969
35. Qu Y, Soininen JP, Nurmi J (2007) A genetic algorithm for scheduling tasks onto dynamically reconfigurable hardware. In: Proceedings of the IEEE international symposium on circuits and systems, pp 161–164
36. Correa R, Ferreira A, Rebreyend P (1999) Scheduling multiprocessor tasks with genetic algorithms. IEEE Trans Parallel Distrib Syst 10(8):825–837
37. Sharma A, Singh N, Hans A, Kumar K (2014) Review of task scheduling algorithms using genetic approach. In: Proceedings of the computational intelligence on power, energy and controls with their impact on humanity (CIPECH). pp 169–172
38. Iranmanesh A, Naji HR (2021) DCHG-TS: a deadline-constrained and cost-effective hybrid genetic algorithm for scientific workflow scheduling in cloud computing. Clust Comput 24(2):667–681
39. Basu S et al (2018) An intelligent/cognitive model of task scheduling for IoT applications in cloud computing environment. Futur Gener Comput Syst 88:254–261
40. Sathish K, RamaMohan Reddy A (2017) Workflow scheduling in grid computing environment using a hybrid gaaco approach. J Inst Eng (India): Ser B 98:121–128