A late-mover genetic algorithm for resource-constrained project-scheduling problems

Liu, Yongping; Huang, Lizhen; Liu, Xiufeng; Ji, Guomin; Cheng, Xu; Onstein, Erling

Published in:
Information Sciences

Link to article, DOI:
10.1016/j.ins.2023.119164

Publication date:
2023

Document Version
Publisher's PDF, also known as Version of record

Information Sciences 642 (2023) 119164

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

A late-mover genetic algorithm for resource-constrained project-scheduling problems

Yongping Liu [a], Lizhen Huang [a,*], Xiufeng Liu [b], Guomin Ji [a], Xu Cheng [a], Erling Onstein [a]

[a] Department of Manufacturing and Civil Engineering, NTNU, Norwegian University of Science and Technology, Teknologivn 22, 2815 Gjøvik, Norway

[b] Department of Technology, Management and Economics, DTU, Technical University of Denmark, Produktionstorvet, Building 424, room 006, 2800 Kgs. Lyngby, Denmark

ARTICLE INFO

Dataset link: https://github.com/ypliu2021/algorithm-for-rcpsp

Keywords:
GA
RCPSP
Heuristic algorithm
Optimization

ABSTRACT

The Resource-Constrained Project Scheduling Problem (RCPSP) plays a critical role in various management applications. Despite its importance, research efforts are still ongoing to improve lower bounds and reduce deviation values. This study aims to develop an innovative and straightforward algorithm for RCPSPs by integrating the "1+1" evolution strategy into a genetic algorithm framework. Unlike most existing studies, the proposed algorithm eliminates the need for parameter tuning and utilizes real-valued numbers and path representation as chromosomes. Consequently, it does not require priority rules to construct a feasible schedule. The algorithm's performance is evaluated using the RCPSP benchmark and compared to alternative algorithms, such as cWSA, Hybrid PSO, and EESHHO. The experimental results demonstrate that the proposed algorithm is competitive, while the exploration capability remains a challenge for further investigation.

1. Introduction

The Resource-Constrained Project Scheduling Problem (RCPSP) is a key issue in many applications, including job-shop scheduling [1,2], construction management scheduling [3,4], and assembly shop scheduling [5,6]. The objective of RCPSPs is to find an optimal schedule that can minimize the total project duration while satisfying resource constraints and precedence constraints among its activities. However, RCPSP is known to be a non-deterministic polynomial-time (NP) hard problem [7]. For decades, researchers have studied methods for obtaining high-quality scheduling solutions within an acceptable time, and this area continues to attract increasing interest.

Recent surveys [8] show that the best exact methods can only solve instances with a maximum of 60 activities without high resource constraints. However, many real-world projects often have more than 60 activities and are subject to various resource constraints. Therefore, many recent research efforts have focused on developing heuristic methods for scheduling optimization. However, finding good lower bound and lower deviation values remains a challenge for many heuristic approaches. One potential reason is that numerous heuristic methods entail varying parameter configurations, necessitating extensive tuning to attain optimal

* Corresponding author.
E-mail addresses: yongping.liu@ntnu.no (Y. Liu), lizhen_huang@ntnu.no (L. Huang), xiuli@dtu.dk (X. Liu), guomin.ji@ntnu.no (G. Ji), xu.cheng@ntnu.no (X. Cheng), erling.onstein@ntnu.no (E. Onstein).

https://doi.org/10.1016/j.ins.2023.119164
Received 18 May 2021; Received in revised form 9 May 2023; Accepted 13 May 2023

Available online 19 May 2023

performance. [9]. For instance, cWSA (combinatoric weighted superposition attraction) [10], hybrid PSO (particle swarm optimization) [11] and EESHHO (Elite Evolution Strategy with Harris Hawks Optimization) [12] are three powerful heuristic algorithms that claimed they can handle medium-scale projects (i.e., a problem with 60 activities), with an average deviation of less than 5%. However, selecting the right combinations of parameters remains a challenge for these three methods. cWSA integrates various operators, including the levy flight mechanism [13,23] and SSGS (serial scheduling generation scheme) [14], making it difficult to select the correct parameters, such as the step size and the length for Levy flight, and the priority rules for the SSGS. Hybrid PSO has five parameters that need tuning [11], including initial weight, two learning factors, swarm size, and peak length. EESHHO uses Gaussian probability distribution as its core operator of EES [12], where different parameters in the distribution can result in different solutions. Therefore, for RCPSP, minimizing the level of deviation while reducing the algorithm's complexity is crucial.

In this paper, we introduce a new genetic algorithm, named *late-mover genetic algorithm (LMGA)*, which aims to reduce the complexity of search methods for RCPSP. The LMGA framework utilizes the worst offspring to compete with its parent generation using a local search optimizer as the source of evolution. To simplify the algorithm, we incorporate a "1+1" evolution strategy, which requires only one individual from the population and one mutation process. Furthermore, we introduce a local search operator into the model, eliminating the need for any priority rules when building a feasible schedule. We also focus on the representation of the chromosome path, which not only simplifies direct operations on a solution, but also enhance its interpretability. With these three strategies, the proposed model is easy to use for real-world problems and requires no parameter tuning. We evaluate the performance of our approach on instances from the project scheduling problem library (PSPLIB),[1] specifically J30, J60, and J120. In summary, Our proposed model provides a novel alternative for solving RCPSP and it makes the following contributions:

- We propose a simple algorithm based on the conception of "1+1" evolutionary strategy and Micro GA. Compared with cWSA, Hybrid PSO and EESHHO, the proposed algorithm is a competitive method for RCPSPs.
- Moreover, the proposed algorithm is a promising method for solving RCPSPs as it eliminates the need for parameter tuning and achieves satisfactory deviation rates. As mentioned previously, parameter tuning is a significant obstacle in optimization, requiring considerable effort for most heuristic methods. In fact, the performance for the same instance can vary greatly, depending on different combinations of parameters, such as population size and probability values used in the heuristics [9]. In contrast, the proposed method has made great progress in this regard.

The remainder of this paper is structured as follows. Section 2 reviews related work; section 3 presents the proposed algorithm; section 4 performs the evaluation; and section 5 concludes the paper and presents the future research directions.

2. Literature review

RCPSP is a well-known problem that has been studied for many years [8]. Three broad categories of methods have been developed to solve RCPSP: exact methods, heuristics, and meta-heuristics. Exact methods include the Program Evaluation and Review Technique (PERT) [15,16], Critical Path Management (CPM) [17], Mixed-Integer Programming (MIP) [18,19] and Branch and Bound (BB) [9]. However, PERT and CPM methods assume unlimited resource availability, which is often unrealistic [20]. Whilst, other methods, such as LIP and BB, can be computationally expensive for problems with many activities [5]. Furthermore, even the best current exact methods can only solve the problems with a maximum of 60 activities and without high resource constraints [8]. Heuristics methods are categorized as either single-pass heuristics (constructive heuristics) or multiple-pass heuristics (improvement heuristics), with Forward-Backward Improvement (FBI) and Neighborhood Search (NS) being their representative methods [9]. In heuristics, Schedule Generation Scheme (SGS) is widely used to create schedules with one or more priority rules, either in series or in parallel. However, it is challenging to find an effective priority rule for a wide range of RCPSP [5]. Meta-heuristic methods, such as genetic algorithms (GA) [21] and difference evolution (DE) [9,20], are also widely used for RCPSPs. However, parameter tuning presents a potential challenge for these methods, as different combinations of parameters, such as population size and probability values, can significantly impact their performance [22,9].

Recently, hybrid meta-heuristic algorithms have been developed to improve the performance of RCPSP. However, the combination of different meta-heuristic methods may not reduce the difficulty of parameter tuning, as different meta-heuristics may require different parameter settings. Table 1 lists several hybrid algorithms that combine different meta-heuristic methods, such as the Consolidated Optimisation Algorithm (COA) [5] and the TS-MODE [9]. Although the hybrid strategy provides a feasible approach to the deficiency problems of evolutionary algorithms, it does not necessarily simplify the parameter tuning process. In fact, the performance of most algorithms is limited to about 10% in J60 with a considerable effort of parameter tuning, which is not ideal for many real-world problems. For instance, COA [5], one of the best performing heuristics for RCPSP, uses hybrid strategies with multiple GA and DE operators, requiring three parameters tuning (CS, PS, and Mrga). Among them, CS has four possible values (10, 25, 50,28 100), PS has four possible values (5, 10, 25, 50), and Mrga has eight possible settings (0.001, 0.01, 0.05, 0.1, adaptive 29 0.05-0.01, adaptive 0.1-0.01, adaptive 0.05-0.001, adaptive 0.1-0.001). It can be time-consuming and difficult to find the optimal combination for these values.

The GA is highly regarded as the first choice for solving complex optimization problems in EA families, as noted by Santiago [31]. However, despite numerous efforts to enhance GA performance through the integration of advanced operators, the challenge of

[1] http://www.om-db.wi.tum.de/psplib.

Table 1
Meta-heuristics algorithms for RCPSP.

Reference	Algorithm	Class	Domain	Hybrid operators	Best average deviation obtained for J30 and J60
Baykasoğlu and Şenol [10]	CWSA	Meta-heuristics	RCPSPs	Levy flight, SSGS, WSA	0.16% and 2.93%
Elsayed et al. [5]	COA	Meta-heuristics	RCPSPs	GA, DE	0% and 10.77%
Sallam et al. [9]	TS-MODE	Meta-heuristics	RCPSPs	DE, FBI, NS	0% and 10.6%
Zamani [24]	ELDRA	Meta-heuristics	RCPSPs	GA, search tree	0% and 10.5%
Goncharov and Leonov [21]	GA	Meta-heuristics	RCPSPs	GA, SGS	10.42% for J60
Paraskevopoulos et al. [25]	SAILS	Meta-heuristics	RCPSPs	Scatter search, adaptive iterated local search	0% and 10.5%
Fang and Wang [26]	SFLA	Meta-heuristics	RCPSPs	SFLA, FBI, SGS	0.18% and 10.66%
Alcaraz and Maroto [27]	GA	Meta-heuristics	RCPSPs	GA, FBC	0.24% for J30
Cheng et al. [20]	FCDE	Meta-heuristics	RCPSPs	DE, SGS, fuzzy c-means clustering	Not mentioned
Tseng and Chen [28]	ANGEL	Meta-heuristics	RCPSPs	GA, ACO, SGS	0.09% and 11.27%
Zheng and Wang [29]	MAOA	Meta-heuristics	RCPSPs	MAOA, FBI, SSGS	0.01% and 10.64%
Shou et al. [11]	PSO	Meta-heuristics	RCPSPs	PSO, SGS	0.4% and 1.33%
Li et al. [12]	EESHHO	Meta-heuristics	RCPSPs	EES, HHO, SGS	0.89% and 3.1%

Algorithm 1: LMGA Algorithm.

Input: Profile of project instances (duration,resource constraints,activity logical constraints)
Output: Best schedule found by the algorithm
1 **Function** `LMGA(`*Profile of project instance*`)`**:**
2 Initialize A as an empty set ;
3 Add a random individual solution to A;
4 Apply `UpdateItemsWithLocalOptimizer` to the items in A (see Algorithm 2);
5 **while** *not Termination Condition* **do**
6 Select the best solution in A ;
7 Apply `Procedure_2` to the selected subset (see Fig. 2);
8 Sort the items in A ;
9 Apply `UpdateItemsWithLocalOptimizer` to the last item in A (see Algorithm 2);
10 Sort the items in A again;
11 Initialize B as an empty set;
12 Add the first and last items of A to B;
13 Convert the first item of A into a date-based schedule and store it as father;
14 Generate a random activity list schedule and store it as mother;
15 Apply either `TwoPointsCrossover` or `OnePointCrossover` to father and mother and store the result as OffspringGroup (see Algorithms 5 and 6);
16 Find the missing activities for each item of OffspringGroup and insert them randomly back;
17 Apply `Make a schedule feasible` to the OffspringGroup (see Algorithm 4);
18 Apply `UpdateItemsWithLocalOptimizer` for items in OffspringGroup (see Algorithm 2);
19 Add updated OffspringGroup to B;
20 Set A equal to B;
21 **return** *the first item of* A;

optimizing its parameters remains. For example, D-GA-MBX [32] has three parameters for tuning with the search space decomposition in GA. IGA-TWD [40] has five parameters for tuning with the time window decomposition method in GA. Additionally, it is interesting to see that Micro-GA, one branch of GA family, is known for its small population size and stress on elitism [33], in which the complexity of the algorithm is significantly reduced. At the same time, it is also seen that the "1+1" evolution strategy is the simplest variant of evolutionary algorithms (EAs) [30], which only requires one individual and one mutation operator. Inspired by these two algorithms, we propose a new algorithm that addresses the challenges of parameter tuning by limiting the population size to one individual and eliminating mutation. As a result, the performance is not influenced by parameter combinations and tuning, allowing for a relatively simple solution finding. However, a strong elitism selection rule from Micro GA may reduce population diversity, leading to premature convergence. To counter this, we use two strategies: we retain the worst and the best offspring for the next iteration during the selection and introduce a new random individual into the population when the last round of selection is completed. This approach guarantees population diversity and is suggested [34,35].

3. The proposed algorithm

The Algorithm 1 describes the proposed algorithm *LMGA*. In this paper, we introduce the concept of the "1 + 1" evolution strategy into the GA algorithm, in which the worst solution competes with the best solution by a local search optimizer.

The algorithm consists of two steps: the front part, including steps from 5 to 12, uses real date value solutions as representation. The back part, including steps from 13 to 15, uses schedule activity solutions as representation (see Algorithm 1). But finally, they are transformed into date real value solution forms, which will be sent back to front part for evolution. In the following, we use an example to illustrate the representation in front part of algorithm, a sequence with nine activities (see the upper table in Fig. 1). We sort the activities according to start dates and obtain a new sequence of activities (the lower table in Fig. 1).

Algorithm 2: UpdateItemsWithLocalOptimizer.

Input: $b4$ (a feasible activity sequence)
Output: $actes$ (a start day value sequence)

Function `UpdateItemsWithLocalOptimizer`($b4$):
  Initialize $actes$ as a zero vector of length *len(Duration)*;
  Initialize $parallel$ and $successor$ as empty sequences;
  for $i \in \{0, \dots, len(Duration)\ -\ 1\}$ **do**
    if $i = 0$ **then**
      $acts_i \leftarrow 0$; *// Set the start day of the first activity to zero*
    else
      Set $tempb4 \leftarrow b4_{[0:i+1]}$; *// Get the subsequence of $b4$ up to index $i+1$*
      Initialize ind and $tempaub$ as empty sequences;
      for $m \in \{0, \dots, len(tempb4)\ -\ 1\}$ **do**
        $ind \leftarrow ind \oplus \{acts_{b4.\text{index}(tempb4_m)}\}$; *// Get the start days of the activities in $tempb4$*
        $tempaub \leftarrow tempaub \oplus \{acts_{b4.\text{index}(tempb4_m)} + Duration_{tempb4_m - 1}\}$; *// Get the end days of the activities in $tempb4$*
      $aub \leftarrow max(tempaub)$; *// Get the maximum end day among the activities in $tempb4$*
      if $b4_{i-1} \in PA_{b4_i - 1}$ **then**
        $successor \leftarrow successor \oplus \{(b4_{i-1}, b4_i)\}$; *// Add a successor relation between $b4_{i-1}$ and $b4_i$*
        $paact \leftarrow tempb4 \cap PA_{b4_i - 1}$; *// Get the sequence of predecessors of $b4_i$ that are also in $tempb4$*
        $paact1 \leftarrow ind + paact.Duration$; *// Get the lower bounds of the predecessors of $b4_i$ as a vector*
        $paalb \leftarrow max(paact1)$; *// Get the maximum lower bound among the predecessors of $b4_i$ as a scalar*
        $ind_i \leftarrow paalb$, and $alb1 \leftarrow paalb$; *// Set the corresponding elements of the vectors* **i** *and* **alb1** *to $paalb$*
      else
        $parallel \leftarrow parallel \oplus \{(b4_{i-1}, b4_i)\}$; *// Add a parallel relation between $b4_{i-1}$ and $b4_i$*
        $tempalb \leftarrow ind + paact.Duration - aub$; *// Get the difference between the lower bounds and the upper bound as a vector*
        $alb \leftarrow 0$ if $\min(tempalb) > 0$ else $\max(tempalb) - \min(tempalb)$; *// Get the minimum difference if it is positive, or the range of differences otherwise*
        $ind_i \leftarrow aub + alb$; *// Set the start day of $b4_i$ as the sum of the upper bound and the difference*
      while `ResourceConstraints`$(ind, b4_{[0:i+1]}, alb_1, aub, rsl) \neq 0$ **do**
        $alb_1 \leftarrow alb_1 + 1$; *// Increase the temporary alb value*
        $ind_i \leftarrow alb_1$; *// Set the current index value to the temporary alb value*
      for $n \in \{0, 1, \dots, len(ind) - 1\}$ **do**
        $acts_n \leftarrow ind_n$; *// Update the start date values vector*
  Return $acts$;

Algorithm 3: Resource Constraints Calculation.

Input: individual, target activity, alb, aub, resource constraints
Output: sum of resource violations for each day in this period

Function `ResourceConstraints`(*individual, targetActivity, alb, aub, resourceConstraints*):
  $T_2 \leftarrow (0, \dots, aub)$; *// Create a sequence of days from 0 to aub*
  $T_4 \leftarrow T_2[alb:]$; *// Subset of days starting from alb*
  sumD $\leftarrow ()$; *// Initialize an empty sequence for storing sum of violations*
  for $i \in T_4$ **do**
    $D \leftarrow ()$; *// Initialize an empty sequence for storing violations*
    for $j = 0$ **to** length(individual) $- 1$ **do**
      $D \leftarrow D \oplus \{(hFunc(i - \text{individual}[j]) - hFunc(i - \text{individual}[j] - \text{duration}[\text{targetActivity}[j] - 1])) * \text{resourceConstraints}[\text{targetActivity}[j] - 1]\}$;
    sumD $\leftarrow$ sumD $\oplus \{\sum D\}$;
  $D_0 \leftarrow (), D_1 \leftarrow (), D_2 \leftarrow (), D_3 \leftarrow ()$;
  forall $k \in$ (sumD $-$ resourceConstraints) **do**
    if $k_1 > 0$ **then**
      $D_0 \leftarrow D_0 \oplus \{k_1\}$;
    if $k_2 > 0$ **then**
      $D_1 \leftarrow D_1 \oplus \{k_2\}$;
    if $k_3 > 0$ **then**
      $D_2 \leftarrow D_2 \oplus \{k_3\}$;
    if $k_4 > 0$ **then**
      $D_3 \leftarrow D_3 \oplus \{k_4\}$;
  $tSum \leftarrow \sum D_0 + \sum D_1 + \sum D_2 + \sum D_3$;
  return $tSum$;

Given a feasible schedule sequence of activities, Procedure 1 is used as a local optimizer to generate the best solution according to the start dates of the activities. This procedure has two steps, namely `Date transfer to schedule` and `Schedule transfer to date`, described as follows:

Activity	1	2	3	4	5	6	7	8	9
Start date	0	0	6	6	8	12	9	15	21

Sort by start date

Activity	1	2	3	4	5	**7**	**6**	8	9
Start date	0	0	6	6	8	9	12	15	21

Fig. 1. An example of schedule sequence.

Procedure 1 Apply local search to the individual whose index is the last one.

```
Step 1: Date transfer to schedule
    1.1: Convert the date plan to a schedule
    1.2: Make the schedule feasible (see \reftext{Algorithm \ref{alg:mga}})
Step 2: Schedule transfer to date
    2.1: Introduce a feasible schedule
    2.2: Make a day plan for the given schedule (see \reftext{Algorithm \ref{alg:localoptimizer}})
```

Algorithm 4: Make a schedule feasible.

Input: an activity sequence x
Output: a feasible activity sequence

Function `MakeScheduleFeasible(`x`):`
  while *True* **do**
    $violation \leftarrow 0$; *// Initialize violation counter*
    for $i \in \{0, \ldots, len(x) - 1\}$ **do**
      if *not list*$(x[i:] \cap PA_{x_i - 1})$ **then**
        continue ; *// Skip to the next iteration if there's no intersection*
      else
        $violation \leftarrow violation + 1$; *// Increment the violation counter*
    if $violation = 0$ **then**
      break ; *// Break the loop if no violations*
    $b \leftarrow x$.copy() ; *// Create a copy of the activity sequence x*
    for $i \in \{0, \ldots, len(x) - 1\}$ **do**
      $intersection \leftarrow$ list$(x[i:] \cap PA_{x_i - 1})$; *// Get the intersection of subsequences*
      if *not intersection* **then**
        continue ; *// Skip to the next iteration if there's no intersection*
      else
        $s1 \leftarrow$ choice$(intersection)$; *// Choose a random element from the intersection*
        $temp \leftarrow x_i$; *// Store the current element in a temporary variable*
        $s2 \leftarrow x$.index$(s1)$; *// Get the index of the randomly chosen element*
        $b_i \leftarrow s1$; *// Update the copied sequence with the chosen element*
        $b_{s2} \leftarrow temp$; *// Update the copied sequence with the previously stored element*
    $x \leftarrow b$; *// Update the activity sequence with the modified copy*
  return x ; *// Return the feasible activity sequence*

In Step 1.1, the schedule sequence is generated by sorting the start dates in ascending order. For example, in Fig. 1, the activities, 1 and 2, have the same start date of 0, which is at the head of the schedule sequence. However, activity 2 is added behind activity 1, since it is larger than activity 1. Activity 7 starts on date 9 which is in the 6th place after the activities are sorted by the start date. Thus, activity 7 is moved one place ahead after the sorting. After sorting, it is possible for activity list to violate the precedence constraints. Therefore, Step 1.2, a correction mechanism, is made to generate a feasible schedule, as described in Algorithm 4.

As shown in Fig. 1, the schedule of activities is determined when the corresponding start dates are sorted. However, the relevant activity schedule sequence may violate the logic precedence relationship, as the start dates are randomly chosen. In Algorithm 4, we check the activity sequence from the top to bottom. During the checking process, if one activity (A) is found its succedent activity (B) stands in front of it, which means there is a violation of precedence constraints, then the positions of A and B will be exchanged and the checking procedure will restart again from the first activity according to the new adjusted list. Then, a feasible schedule sequence that satisfies the dependent constraints among activities can be achieved through our checking process. The relevant information can be seen in the Algorithm 4.

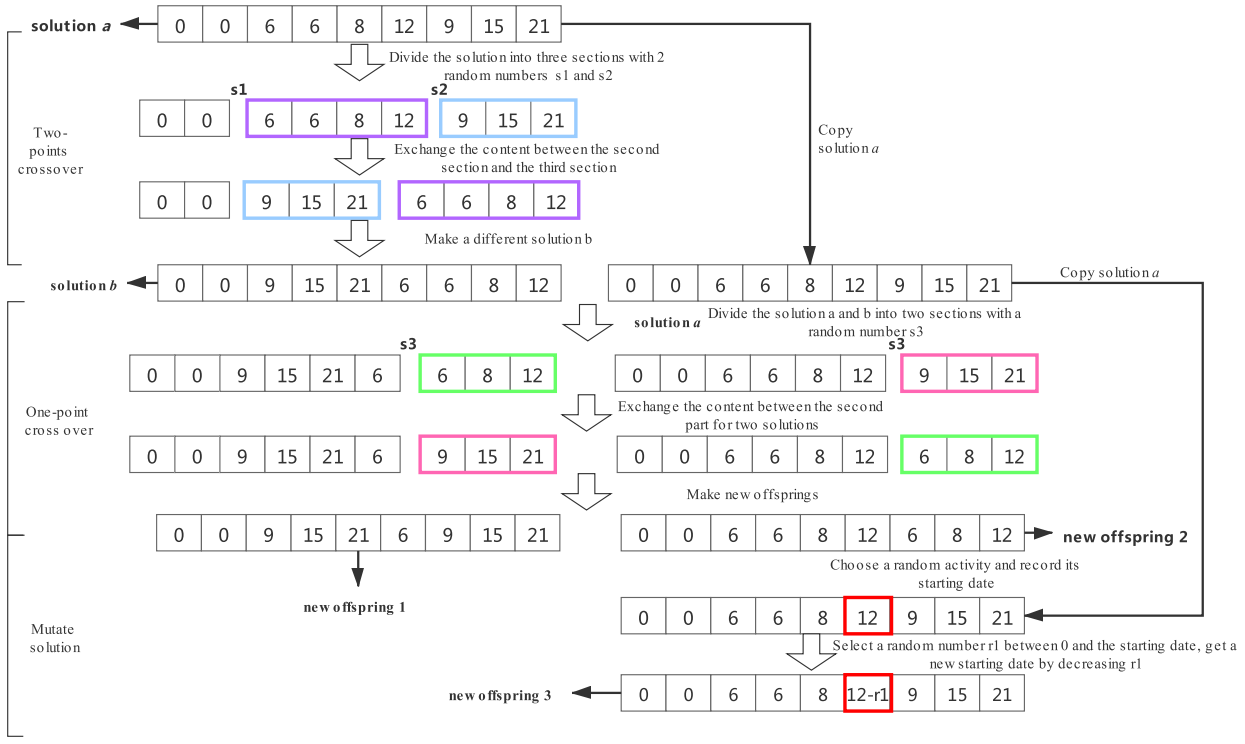


Fig. 2. Procedure 2: GA crossover.

We now address the resource constraints by the process of "schedule transfer to date". Step 2 (`schedule transfer to date`) in Procedure 1 is essentially a local optimization process, which is further described in Algorithm 2 (Make a day plan for the given schedule in "schedule transfer to date" in Procedure 1). The process to check "`whether resource constraints are violated?`" (see the step 25 in Algorithm 2) is the key to solving the resource constraint problem. The relevant details can be seen in the Algorithm 3 (Resource Constraints calculation), verified by the following two equations. The first is called the heavy side function (see Equation (1)). If the independent variable is positive, then the function will generate output 1, otherwise 0.

$$f(x) = \begin{cases} 1 & \text{if } \exists x \geqslant 0 \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

The second function is used to calculate the computing resource j for all activities at day t (see Equation (2)):

$$R_j(t) = \sum_{i=1}^{n} r_{ij} * \left[f\left(t - s_i\right) - f\left(t - s_i - d_i\right) \right] \tag{2}$$

where $f(\cdot)$ is noted as the heavy side function; s_i is the start date of activity i; d_i is the duration of activity i; r_{ij} is the computing resource j required for the activity i; and $R_j(t)$ is the total resource j within an available limit at date t. As the above two equations have been validated for resource calculation, e.g., construction management [36], we just follow these strategies. When the resource constraints are violated, an additional penalty is given to the makespan for fitness evaluation. In Algorithm 2, given the setting date for the plan, if a penalty is applied after checking, the algorithm will increase one day for activity until the resource constraints are met.

Therefore, in Procedure 1, our algorithm can provide the local best date plan that meets not only the resource constraints, but also the logic precedence relationship of activities. Unlike traditional SGS [8], our algorithm does not require priority rules to create a feasible schedule.

We now describe Procedure 2 shown in Algorithm 1. This procedure is used to generate offspring through the crossover described in Fig. 2. In contrast to traditional GA in which the mutation occurs after the crossover, the proposed method directly executes the crossover operator on the parent generation to obtain an offspring. At the end of front part of Algorithm 1, the best and worst solutions are selected from the individuals generated by mixing parents and their offspring. The two-point crossover in Fig. 2 is included to produce a similar offspring b as from the solution a. As the activity list schedule sequence is determined by the start dates, a new date real value schedule sequence can be generated if the logic precedence relationship of the start dates is changed. By crossover, most of the sections in the parent schedule sequence are kept, while only a few sections are adjusted. However, the resulting offspring need further optimization by a local search operation, as they may not satisfy the time and resource constraints (see Procedure 1).

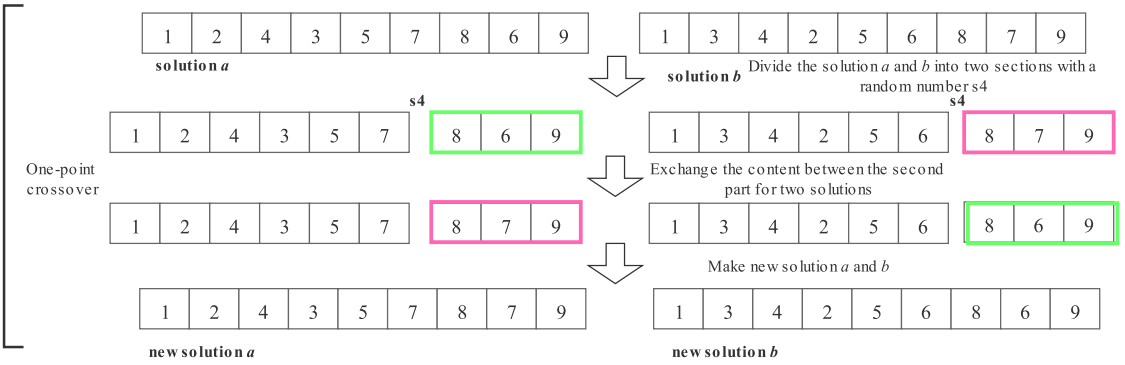


Fig. 3. Procedure 3: GA crossover.

The two-point and one-point crossover in Procedure 2 are shown in the following:

Algorithm 5: Two-points crossover.

Input: Two solutions a and b
Output: A new offspring c
Function `TwoPointsCrossover(a, b)`:
  Choose two random numbers i and j such that $1 \le i < j \le n$, where n is the length of the solutions ; *// Select two random indices*
  Divide each solution into three sections: a_1, a_2, a_3 and b_1, b_2, b_3 ; *// Split solutions into sections*
  $c \leftarrow a_1 \oplus b_2 \oplus a_3$; *// Create offspring by concatenating sections*
  return c ; *// Return the new offspring*

A new offspring is created by combining solution a and solution b with a one-point crossover, described as follows:

Algorithm 6: One-point crossover.

Input: Two solutions a and b
Output: A new offspring c
Function `OnePointCrossover(a, b)`:
  Choose a random number i such that $1 \le i < n$, where n is the length of the solutions ; *// Select a random index*
  Divide each solution into two sections: a_1, a_2 and b_1, b_2 ; *// Split solutions into sections*
  $c \leftarrow a_1 \oplus b_2$; *// Create offspring by concatenating sections*
  return c ; *// Return the new offspring*

Inspired by the mutation approach set out in [37], we generate another new offspring at the last step in Procedure 2 directly from the mutation of the parent generation, in which it selects a random activity and reduces it by a random number. Details are as follows:

Algorithm 7: Mutation.

Input: A solution a
Output: A new offspring c
Function `Mutation(a)`:
  Choose a random activity k and record its start date s_k ; *// Select a random activity*
  Choose a random number d between 0 and s_k ; *// Select a random date shift*
  $c \leftarrow a$; *// Initialize offspring with the original solution*
  $c_k \leftarrow s_k - d$; *// Update the start date of activity k*
  return c ; *// Return the new offspring*

When a temporary best solution has been generated by the front part of our algorithm, the back part of algorithm will produce two feasible schedule sequences using Procedure 3, as shown in Fig. 3.

Procedure 3 performs the one-point crossover, which is the same as the crossover in Procedure 2. It should be noted that here the chromosome is a schedule activity sequence. In Procedure 3, two new active schedules are generated (see Fig. 3). However, since the generated schedules may not follow the constraints of the logical relationships, they must be corrected to feasible schedules using the procedure step 16 and 17 in Algorithm 1. Two offspring schedules are generated by the traditional GA one-point crossover process, which will be sent back to the front part, optimized by the local search operation of Procedure 1.

4. Experiments

4.1. Experimental settings

We conducted the experiments on a laptop with a 1.90 GHz Core (TM) i7-8650 CPU, 16 GB RAM and Windows 10 installed. All programs were implemented with Python 3.6. We use PSPLIB for benchmarking with the problems of J30, J60 and J120, which correspond to 480, 480 and 600 instances. An instance can have 30, 60 or 120 activities.

For all tests, we use a population size of 1 and set the probability of crossover to 1 to save computing resources. We design the following three test cases according to the problems and termination conditions.

- **Test 1**: Only J30 and J60 were tested. For J60, we record the makespan (i.e., the duration between start and end times of a job) of each instance with 30 iterations.
- **Test 2**: Only J30 and J60 were tested. Test 2 aims to check the consistency and exploration ability of the proposed algorithm. Therefore, there is a small adjustment for the termination here. The test will end as long as either of the following two conditions is met: 1) the best makespan is found; or 2) the best makespan is not found, but the iteration reaches 60.
- **Test 3**: Only J120 was tested. All instances were explored. The test ends when the best makespan is found. Given the condition that the best makespan is not found, if the number of iterations is more than 15, then the test ends.

The deviation (noted as *dev*) is used to measure the quality of the algorithm [38] defined as follows:

$$dev = \frac{F - LB}{LB} \tag{3}$$

where F is the makespan of an instance found by our algorithm and LB is the low-bound optima given by PSPLIB. The optima is obtained from [39].[2]

Therefore, the average deviation (noted as *adev*) for each problem can be defined as:

$$adev = \frac{\sum_{i=1}^{n} dev_i}{n} \times 100\% \tag{4}$$

where dev_i is the deviation of an instance, i, and n is the number of instances of a problem.

4.2. Results and analysis

This section will use the PSPLIB benchmark to evaluate the proposed algorithm, namely J30, J60 and J120, and then compare them with state-of-the-art algorithms. Note that the following experimental results and relevant supplement files are stored at *https://github.com/ypliu2021/algorithm-for-rcpsp*.

4.2.1. Results for J30 sets

Fig. 4 presents a summary of the results from two tests conducted on all 480 instances within the J30 sets, which are categorized into eight groups. In Test 1, the largest group contains instances with a deviation less than or equal to 0%, accounting for 394 instances or 82% of the total. This group's cumulative probability is also 82%, indicating an 82% chance of reaching the optimum solution in a single experiment.

The following three groups consist of instances with deviations between 1% and 2%, 2% and 3%, and 3% and 4%, comprising 41, 20, and 13 instances, respectively. When considering a deviation below 4%, the cumulative probability rises to 97.5%, suggesting a 97.5% likelihood of reaching the optimum solution with a deviation under 4% in a single experiment.

Interestingly, there are only 12 instances exhibiting deviations between 4% and 7%, with no instances exceeding 7%. The average deviation is a mere 0.453%, while the average time spent on each instance is 5.06 seconds. Detailed information on the solutions can be found in the supplementary materials (refer to Supplement 1: Results for J30 in Test 1).

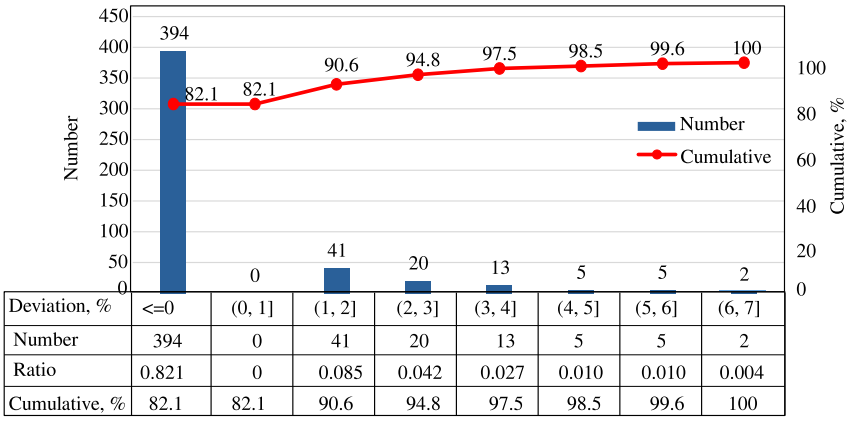
In comparison to Test 1, the cumulative probability distribution curve for Test 2 reveals a slight improvement in the search results, as illustrated in Fig. 4(b). The average deviation is reduced to 0.355%, while the average CPU time increases to 9.49 seconds. As anticipated, a greater number of instances (413) are discovered in Test 2 with the optimal lower bound, leading to a reduction in the count of middle groups. Further details on the solutions can be found in the supplementary materials (refer to Supplement 3: Results for J30 in Test 2).

Based on the aforementioned results, it can be safely concluded that the proposed algorithm for J30 exhibits consistent performance. The average deviation for this algorithm is approximately 0.4%, with an average time consumption of 7.5 seconds.

4.2.2. Results for J60 sets

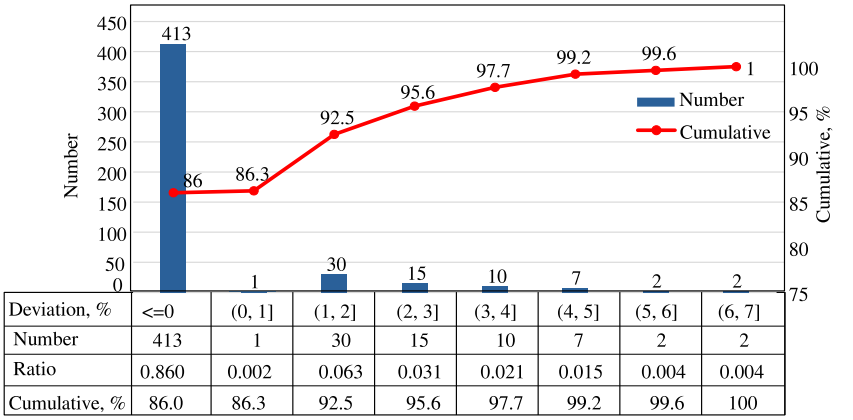
Fig. 5(a) displays the experimental results of Test 1 for the J60 sets, which are organized into 23 groups according to deviations ranging from 0% to 22%. The first group, with deviations less than or equal to 0%, encompasses the majority of instances, totaling

[2] The optimum comes from https://ptal.github.io/scheduling-data.html, in which the last date was 29 August, 2019.



Deviation, %	<=0	(0, 1]	(1, 2]	(2, 3]	(3, 4]	(4, 5]	(5, 6]	(6, 7]
Number	394	0	41	20	13	5	5	2
Ratio	0.821	0	0.085	0.042	0.027	0.010	0.010	0.004
Cumulative, %	82.1	82.1	90.6	94.8	97.5	98.5	99.6	100

(a) Test 1 for J30



Deviation, %	<=0	(0, 1]	(1, 2]	(2, 3]	(3, 4]	(4, 5]	(5, 6]	(6, 7]
Number	413	1	30	15	10	7	2	2
Ratio	0.860	0.002	0.063	0.031	0.021	0.015	0.004	0.004
Cumulative, %	86.0	86.3	92.5	95.6	97.7	99.2	99.6	100

(b) Test 2 for J30

Fig. 4. Experimental deviation frequency and cumulative probability distribution for J30 sets.

346 or 72% of the 480 instances. In contrast, the other groups contain significantly fewer instances. For instance, the second-largest group comprises only 20 instances with deviations between 1% and 2%.

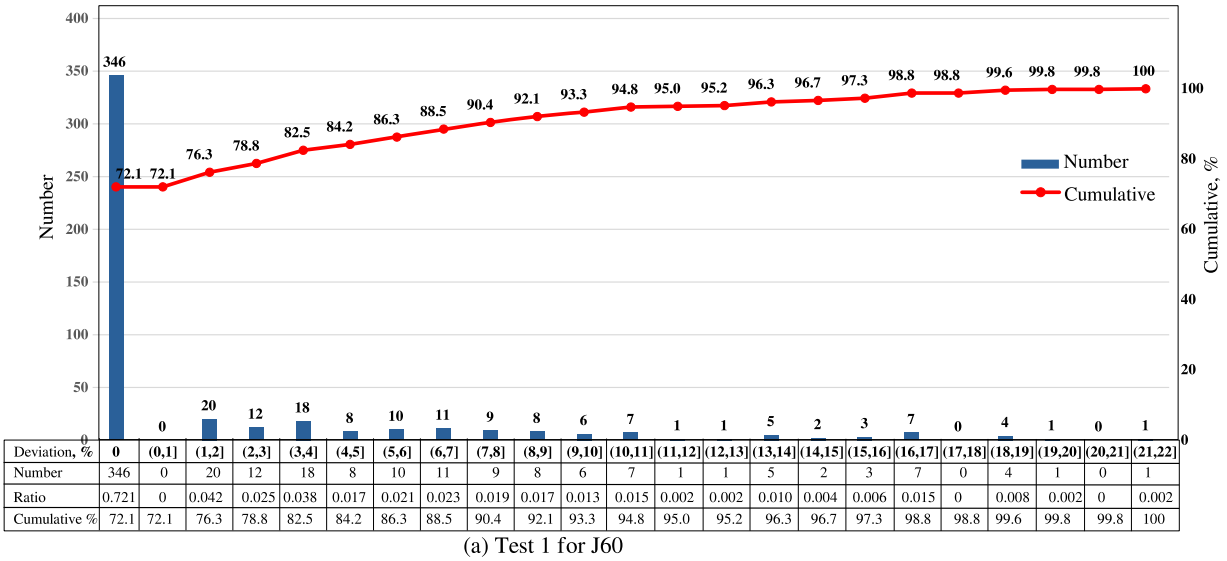
A notable observation is the presence of 38 instances, classified as hard cases, with deviations between 9% and 22%. These instances account for 8% of the total 480 instances. The cumulative probability curves indicate that in a single experiment, 93.34% of instances achieve a fitness value with a deviation of less than 10%. Only 32 instances remain unsolved using the proposed method. With an average deviation of 1.99% and an average CPU time of 13.35 seconds, the proposed method demonstrates well handling of the J60 sets. Further details on the solutions can be found in the supplementary materials (refer to Supplement 2: Results for J60 in Test 1).

Fig. 5(b) presents the results of Test 2, revealing two subtle changes in comparison to Test 1. First, in the initial group with deviations less than or equal to 0%, 351 instances are found to have low-bound optima (see Supplement 4: Results for J60 in Test 2). Second, the remaining groups exhibit a tendency towards a left-skewed distribution. For instance, the number of instances in the third group, with a variance between 2% and 3%, increases from 12 to 17. Although the starting point of Test 2 has risen by 1.45%, the cumulative probability distribution remains largely unchanged. This indicates that our algorithm consistently manages medium-sized problems, such as the J60 set. Notably, in Test 2, the average deviation is 1.74%, and the average CPU time is 29 seconds. These results align with expectations, as more time was devoted to exploring optima in Test 2. Consequently, based on the findings from both tests, we can deduce a final deviation of 1.87% and an average CPU time of 22 seconds.

4.2.3. Results for large-scale schedule

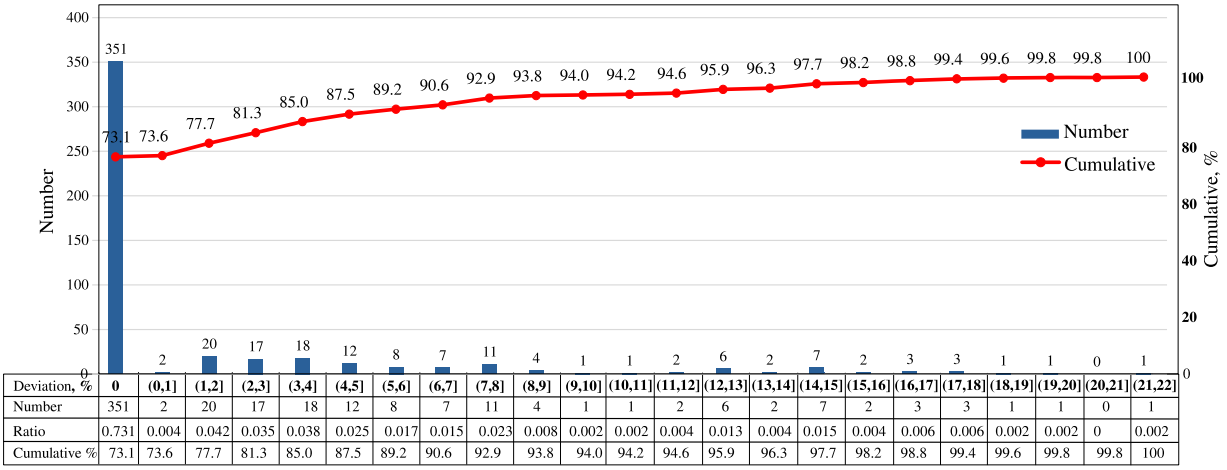
Fig. 6 provides a summary of the experimental results for the large-scale J120 sets, consisting of 600 instances. Three prominent peaks are evident in the data. The first peak represents the largest group with a deviation of 0%, featuring 168 instances solved at optimal values. The second peak comprises 43 instances, exhibiting a deviation between 9% and 10%. The third peak encompasses two groups: 34 instances with a deviation between 7% and 8%, and 33 instances with a deviation between 13% and 14%.

The cumulative probability curve demonstrates that 554 instances have a deviation below 16%, signifying a 90.83% probability of attaining a fitness value in a single experiment. Among the groups with a deviation between 0% and 16%, an average of 25 instances displays a deviation of 8.96%. The relevant standard deviation is 9.5%, indicating that the number of instances lies within two standard deviations from the mean—specifically, within the range of (25 - 2 × 9.5, 25 + 2 × 9.5). Furthermore, 46 instances



Deviation, %	0	(0,1]	(1,2]	(2,3]	(3,4]	(4,5]	(5,6]	(6,7]	(7,8]	(8,9]	(9,10]	(10,11]	(11,12]	(12,13]	(13,14]	(14,15]	(15,16]	(16,17]	(17,18]	(18,19]	(19,20]	(20,21]	(21,22]
Number	346	0	20	12	18	8	10	11	9	8	6	7	1	1	5	2	3	7	0	4	1	0	1
Ratio	0.721	0	0.042	0.025	0.038	0.017	0.021	0.023	0.019	0.017	0.013	0.015	0.002	0.002	0.010	0.004	0.006	0.015	0	0.008	0.002	0	0.002
Cumulative %	72.1	72.1	76.3	78.8	82.5	84.2	86.3	88.5	90.4	92.1	93.3	94.8	95.0	95.2	96.3	96.7	97.3	98.8	98.8	99.6	99.8	99.8	100

(a) Test 1 for J60



Deviation, %	0	(0,1]	(1,2]	(2,3]	(3,4]	(4,5]	(5,6]	(6,7]	(7,8]	(8,9]	(9,10]	(10,11]	(11,12]	(12,13]	(13,14]	(14,15]	(15,16]	(16,17]	(17,18]	(18,19]	(19,20]	(20,21]	(21,22]
Number	351	2	20	17	18	12	8	7	11	4	1	1	2	6	2	7	2	3	3	1	1	0	1
Ratio	0.731	0.004	0.042	0.035	0.038	0.025	0.017	0.015	0.023	0.008	0.002	0.002	0.004	0.013	0.004	0.015	0.004	0.006	0.006	0.002	0.002	0	0.002
Cumulative %	73.1	73.6	77.7	81.3	85.0	87.5	89.2	90.6	92.9	93.8	94.0	94.2	94.6	95.9	96.3	97.7	98.2	98.8	99.4	99.6	99.8	99.8	100

(b) Test 2 for J60

Fig. 5. Experimental deviation frequency and cumulative probability distribution for J60 sets.

Table 2
Average deviation values obtained by GA and other state-of-the-art algorithms for J30.

Reference	Algorithm	Schedules		
		<1,000	1,000	5,000
Elsaved et al. [5]	COA		0.04(%)	
Hua et al. [40]	**IGA-TWD**		**0.24(%)**	**0.06(%)**
Liu et al. [32]	**DGA-DJ**		**0.23(%)**	**0.08(%)**
This study	**LMGA**	**0.4(%)**		
Sallam et al. [9]	TS-MODE		0.06(%)	
Paraskevopoulos et al. [25]	GH+SS(LS)		0.03(%)	
Baykasoğlu and Şenol [10]	CWSA			0.16(%)

exhibit a deviation above 16%. In conclusion, the average deviation for this test is 6.8%, suggesting a 50% probability of finding a solution with a deviation between 6% and 7% in a one-time experiment. The average usage time amounts to approximately 28.85 seconds.

4.3. Comparison with state-of-the-art algorithms

RCPSP is a classic problem and many algorithms have been investigated for it since 1995 [8]. We choose the algorithms listed in Tables 2 and 3 for comparison, because of their good performance in terms of deviation. In Table 3, AD denotes the average

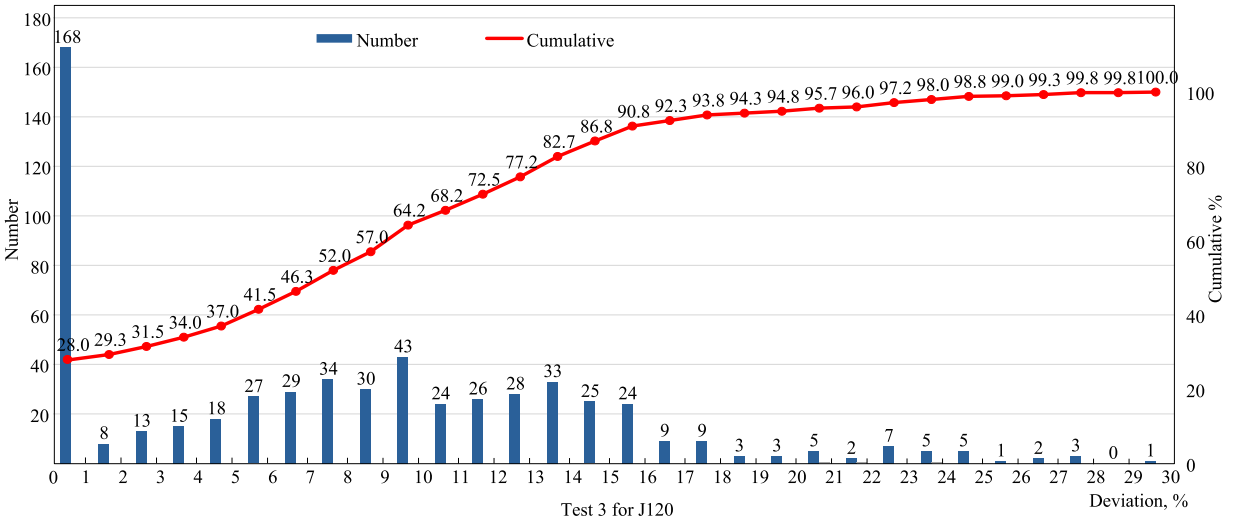


Fig. 6. Experimental deviation frequency and cumulative probability distribution for J120 sets.

Table 3
Comparisons between EESHHO, Hybrid PSO, cWSA, and GA (this study) for RCPSPs.

Reference	Algorithm	Schedules	J30				J60				J120			
			AD	MD	PO	PB	AD	MD	PO	PB	AD	MD	PO	PB
This study	**LMGA**	**1,000 (J30, J60) 1,000 (J120)**	**0.4%**	**7%**	**84%**	**0%**	**1.87%**	**22%**	**72.5%**	**0%**	**6.8%**	**29%**	**28% (168/600)**	**0%**
Shou et al. [11]	Hybrid PSO	5000	0.40%	6.78%	83.96%	0.00%	1.33%	10.34%	71.88%	0.00%	6.4%3	17.53%	31.67%	0.00%
Baykasoğlu and Şenol [10]	cWSA	5000	0.16%		91.6% (440/480)		2.93%		68.33% (328/480)		14.72%		16.83% (101/600)	
Li et al. [12]	EESHHO	5000	0.89%		74.58% (358/480)		3.1%				11.5%		15% (90/600)	

deviation of all instances, while MD is the maximum deviation. A negative deviation means a shorter project duration. PO represents the percentage of instances for which the procedure has found optimal or best-known solutions. PB represents the percentage of instances for which the procedure has found solutions better than the benchmark. In Table 2, COA was developed by Elsayed et al. [5] for the J30 set, with a low deviation of 0.04%. According to [8], GH+SS(LS) is the best algorithm for J30 RCPSP, with a deviation of only 0.03%. COA ranks second according to the deviation.

Compared with the algorithm mentioned above, from the perspective of deviation, our algorithm still needs to expand its searching capacity. The distance is not so far. In fact, the average deviation of our algorithm under J30 is only 0.4%, but we expect that it can be smaller for 5,000 schedules, because it succeeds in finding the optima for 82% of the instances in the J30 set. Moreover, it should be noted that the performances of above mentioned algorithms are achieved by using considerable parameter tuning. In contrast with them, our algorithm is easy to use without the need of parameter tuning.

Table 3 compares our algorithm with three competitive alternatives: cWSA [10], Hybrid PSO [11], and EESHHO [12]. Among these, cWSA optimally solves 440 of the 480 problems [10], outperforming both Hybrid PSO and EESHHO. Our algorithm demonstrates comparable results when handling the J30 set, as depicted in Fig. 4. Specifically, our method achieves optima for 394 and 413 instances in Test 1 and 2, respectively (refer to Supplement 1 for Test 1 and Supplement 3 for Test 2). In comparison to Hybrid PSO, cWSA, and EESHHO, our algorithm obtains similar results without the need for parameter tuning, as shown in Fig. 4. This highlights the competitive performance of the proposed method when dealing with J30 instances.

Table 4 gives the results for the J60 set. According to the survey in [8], GA (FBI) is ranked as the best for the J60 set, with the lowest deviation of 10.42% per 5,000 experiments. The other three algorithms have an average deviation of less than 3%, including cWSA [10], Hybrid PSO [11] and EESHHO [12] (see Table 3). Hybrid PSO is the best, with a deviation of 1.33%. However, it should be noted that this value is obtained with different bases. COA, TS-MODE, GA(FBI) are calculated based on the CPM standard, which is stricter than the base used by cWSA, EESHHO, and Hybrid PSO. Therefore, it is difficult to compare their performance between these two groups. Since our base is similar to that of cWSA, EESHHO and Hybrid PSO, it is possible to compare ours with them. In fact, our algorithm is comparable to cWSA and Hybrid PSO (see Table 3), e.g., the PO ratio with 346 and 351 out of 480 instances optimally solved for Tests 1 and 2, respectively (see Fig. 5). Based on the above results, we can conclude that the proposed algorithm is competitive with these three algorithms, especially considering that the results are obtained without parameter tuning.

Table 4
Average deviation values obtained by LMGA and other state-of-the-art algorithms for J60.

Reference	Algorithm	Schedules		comment
		1,000	5,000	
Elsayed et al. [5]	COA		10.581(%)	CPM base
Sallam et al. [9]	TS-MODE		10.629(%)	CPM base
Goncharov and Leonov [12]	GA(FBI)		10.42(%)	CPM base
This study	LMGA	1.87(%)		benchmark base
Baykasoğlu and Şenol [10]	CWSA		2.93%	benchmark base
Shou et al. [11]	Hybrid PSO		1.33%	benchmark base
Li et al. [12]	EESHHO		3.1%	benchmark base

Table 5
Average deviation values obtained by LMGA and other state-of-the-art algorithms for J120.

Reference	Algorithm	Schedules		comment
		1,000	5,000	
Elsayed et al. [5]	COA		31.19(%)	CPM base
Sallam et al. [9]	TS-MODE		30.59(%)	CPM base
Baykasoğlu and Şenol [10]	CWSA		14.72%	benchmark base
Shou et al. [11]	Hybrid PSO		6.43%	benchmark base
Li et al. [12]	EESHHO		11.5%	benchmark base
This study	**LMGA**	**6.8(%)**		benchmark base

For large-scale projects (J120), the survey in [8] shows that most algorithms can achieve a minimum deviation of 30.45% after 50,000 experiments, including COA and TS-MODE. cWSA, Hybrid PSO and EESHHO can do even better, with deviations as low as about 10% (see Table 5). However, this is because they use different baselines for comparison. In order to compare them on equal footing, the targeted algorithm for comparison here will be cWSA, Hybrid PSO, and EESHHO. Hybrid PSO is the best in terms of deviation, which is 6.43% (see Table 5), and ours is comparable, which is 6.8%. It is better than CWSA (11.5%) and EESHHO (14.72%). In terms of PO ration, Hybrid PSO also ranks best (see Table 3), up to 31.67%, while the other two are around 16%. Ours can be ranked second according to the PO ratio (28%) and the average deviation (6.87%). Note that these values were obtained using 1,000 instances. As mentioned earlier, our average deviation can be smaller with more iterations; for example, about 30/% of the J120 set can find the low-bound optima with our algorithm. In particular, the results are obtained without parameter tuning, which is a significant improvement. Based on these results, we can also conclude that our algorithm is a competitive method, compared with cWSA, Hybrid PSO, and EESHHO in solving large-scale project optimization problems.

4.4. Discussion

Our method diverges from most existing studies in two aspects. Firstly, the number of experimental schedules is within 1,000 schedules in the J30 and J60 sets, respectively. In contrast, many studies use 5,000 or even 50,000 schedules to assess algorithm consistency. We argue that testing with such a large number of instances is of limited use for the results. For the J30 or J60 set, due to the premature character of the heuristic algorithms, the deviation distribution is determined when 1,000 or 5,000 schedules are applied (see Figs. 4 and 5 for deviation results). Consequently, increasing the number of experiments will not significantly alter the distribution, as evidenced by the minimal alterations in the cumulative probability curve illustrated in Fig. 4. Furthermore, as shown in Figs. 4 and 5, the complexity of solving instances in the J30 and J60 sets is different for a given method, since the number of difficult instances is different in the two sets, i.e., about 12% and 34% in the J30 and J60 sets, respectively (see Figs. 4a and 5a). These findings suggest that utilizing 50,000 schedules to evaluate the algorithm's performance is not an optimal option, as the heuristic method relies on random choices. There exists the possibility of getting the optimum solution, although this probability can be extremely low. As a result, the performance of the algorithm is likely to be biased towards easier instances. Secondly, our approach is distinct from other algorithms in that it employs real-valued solutions and path representations as chromosomes, enabling direct operation on instances without the need for priority rules to construct a feasible schedule. This feature is particularly advantageous, as Elsayed et al. have previously highlighted the challenge of identifying an efficient priority rule that performs well across a broad range of RCPSPs [5].

Additionally, the proposed method employs an evolutionary algorithm that utilizes a uniform random choice in the crossover process to generate new offspring. As shown in Figs. 2 and 3, the random choice yields one or two random integers for the positions. This random sampling can be regarded as uncertain input to the algorithm, potentially leading to varying random schedules, and consequently, uncertainty in the resulting schedules. Hence, the generated offspring schedules may not outperform the parent schedule, causing a slower convergence speed due to the increased time spent on the inefficient track-and-error process. To date, many methods are available to mitigate this uncertainty, e.g., [41–43]. Since the relationship between the random choice outcomes and their fitness values can be viewed as a Monte Carlo (MC) experiment, methods such as MC dropout, Markov chain Monte Carlo

techniques, and Bayesian inference can be effective ways to minimize the issue of low sampling efficiency. However, as uncertainty lies beyond the scope of this paper, we will explore this aspect in future work.

Furthermore, the proposed method primarily employs the local search operations to solve RCPSP without parameter tuning (see Procedure 1). However, numerous algorithms are designed to tackle problems across a wide range of application domains, such as hybrid PSO for RCPSP [11], health problems [44,45], and mixed-variable problems [46]. This results in the significant challenge of tuning parameters for different specific domains. Consequently, the adaptive parameter tuning (APT) strategy was introduced in the hybrid PSO [46], which heavily relies on the Cauchy distribution and the Gaussian distribution with fixed variances. In fact, this issue is well-known for most new meta-heuristic algorithms. For example, the Whale Optimization Algorithm (WOA) claims to have fewer and simpler tuning parameters [47], but fine-tuning the step size becomes critical and significantly affects its performance. The Gray Wolf algorithm (GWO), developed based on the leadership hierarchy of gray wolves and their hunting strategies, also faces performance issue due to the control vector. As a result, numerous variants of the GWO algorithm have emerged, such as the Grouped Gray Wolf Optimizer (GGWO) and the Enhanced Gray Wolf Optimizer (IGWO), which incorporate different parameter tuning operators. In contrast, our algorithm streamlines the process by integrating the “1+1” strategy into a Genetic Algorithm framework, eliminating the need for parameter tuning. This enhancement makes our method significantly more user-friendly.

5. Conclusions

RCPSP is a common challenge for many applications. Most existing algorithms require significant effort in parameter tuning to achieve satisfactory results. In this paper, we propose a new method for RCPSP by incorporating a “1+1” evolutionary strategy into the genetic algorithm framework. This proposed algorithm requires only a single population size and employs a 100% mutation probability, thereby eliminating the need for laborious parameter tuning and simplifying its usage. Additionally, our approach utilizes both real-valued solutions and path representations as chromosomes, eliminating the need for priority rules to establish a feasible schedule. We conducted a comprehensive evaluation of the proposed algorithm using J30, J60, and J120 instances, and compared its performance with high-performing algorithms such as hybrid PSO, cWSA, and EESHHO. The results demonstrate that our proposed algorithm is particularly simple and promising for solving RCPSPs. This study represents an attempt to address RCPSPs without parameter tuning while maintaining good accuracy. However, enhancing the search capability of the proposed algorithm remains a challenge, which will be left for future work.

CRediT authorship contribution statement

Conceptualization, **Y.L.** & **L.H**; methodology, **Y.L.**; investigation and resources, **Y.L.**, **L.H.**; writing-original draft preparation, **Y.L.** and **X.L.**; writing-review and editing, ALL; supervision, **Y.L.** and **L.H.**; project administration, **Y.L.** and **L.H.**; funding acquisition, **L.H**. All authors have read and agreed to submit the manuscript.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

https://github.com/ypliu2021/algorithm-for-rcpsp.

Acknowledgements

This research was supported by two projects: Reinforcing the Health Data Infrastructure in Mobility and Assurance through Data Democratization (288856) and circle Wood (328698), funded by the Norwegian Research Council. It also received support from the Flexible Energy Denmark project (8090-00069B) funded by the Innovation Fund Denmark.

Appendix A. Supplementary material

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.ins.2023.119164.

References

[1] M. Thenarasu, K. Rameshkumar, J. Rousseau, S.P. Anbuudayasankar, Development and analysis of priority decision rules using MCDM approach for a flexible job shop scheduling: a simulation study, Simul. Model. Pract. Theory 114 (2022) 102416.
[2] R. Chen, B. Yang, S. Li, S. Wang, A self-learning genetic algorithm based on reinforcement learning for flexible job-shop scheduling problem, Comput. Ind. Eng. 149 (2020) 106778.
[3] J. Liu, Y. Liu, Y. Shi, J. Li, Solving resource-constrained project scheduling problem via genetic algorithm, J. Comput. Civ. Eng. 34 (2) (2020) 04019055.
[4] G. Selvam, T.C. Tadepalli, Genetic algorithm based optimization for resource leveling problem with precedence constrained scheduling, Int. J. Constr. Manag. 1 (10) (2019).

[5] S. Elsayed, R. Sarker, T. Ray, C. Coello, Consolidated optimization algorithm for resource-constrained project scheduling problems, Inf. Sci. 418 (2017) 346–362.
[6] D. Arkhipov, O. Battaïa, A. Lazarev, An efficient pseudo-polynomial algorithm for finding a lower bound on the makespan for the Resource Constrained Project Scheduling Problem, Eur. J. Oper. Res. 275 (2019) 35–44.
[7] J. Blazewicz, J. Lenstra, A. Kan, Scheduling subject to resource constraints: classification and complexity, Discrete Appl. Math. 5 (1983) 11–24.
[8] R. Pellerin, N. Perrier, F. Berthaut, A survey of hybrid metaheuristics for the resource-constrained project scheduling problem, Eur. J. Oper. Res. 280 (2020) 395–416.
[9] K. Sallam, R. Chakrabortty, M. Ryan, A two-stage multi-operator differential evolution algorithm for solving Resource Constrained Project Scheduling problems, Future Gener. Comput. Syst. 108 (2020) 432–444.
[10] A. Baykasoğlu, M. Şenol, Weighted superposition attraction algorithm for combinatorial optimization, Expert Syst. Appl. 138 (2019) 112792.
[11] Y. Shou, Y. Li, C. Lai, Hybrid particle swarm optimization for preemptive resource-constrained project scheduling, Neurocomputing 148 (2015) 122–128.
[12] C. Li, J. Li, H. Chen, A. Heidari, Memetic Harris Hawks Optimization: developments and perspectives on project scheduling and QoS-aware web service composition, Expert Syst. Appl. 171 (2012) 114529.
[13] W. Kaidi, M. Khishe, M. Mohammadi, Dynamic Levy flight chimp optimization, Knowl.-Based Syst. 235 (2022) 107625.
[14] G. Ulusoy, Ö. Hazır, Resource constrained project scheduling, in: An Introduction to Project Modeling and Planning, 2021, pp. 199–250.
[15] I.A. Kartini, T. Aspiranti, A.M. Rani, Implementation of program evaluation and review technique (Pert) to optimize shophouse development projects, J. Manag. Energy Bus. 1 (1) (2021).
[16] E. Akan, S. Bayar, Interval type-2 fuzzy program evaluation and review technique for project management in shipbuilding, Ships Offshore Struct. 17 (8) (2022) 1872–1890.
[17] D. Sarkar, K.N. Jha, S. Patel, Critical chain project management for a highway construction project with a focus on theory of constraints, Int. J. Constr. Manag. 21 (2) (2021) 194–207.
[18] D. Ioan, I. Prodan, S. Olaru, F. Stoican, S.I. Niculescu, Mixed-integer programming in motion planning, Annu. Rev. Control 51 (2021) 65–87.
[19] A. Ham, Dial-a-ride problem: mixed integer programming revisited and constraint programming proposed, Eng. Optim. (2021) 1–14.
[20] M. Cheng, D. Tran, Y. Wu, Using a fuzzy clustering chaotic-based differential evolution with serial method to solve resource-constrained project scheduling problems, Autom. Constr. 37 (2014) 88–97.
[21] E. Goncharov, V. Leonov, Genetic algorithm for the resource-constrained project scheduling problem, Autom. Remote Control 78 (2017) 1101–1114.
[22] A. Ezugwu, D. Prayogo, Symbiotic organisms search algorithm: theory, recent advances and applications, Expert Syst. Appl. 119 (2019) 184–209.
[23] E. Emary, H. Zawbaa, M. Sharawi, Impact of Lèvy flight on modern meta-heuristic optimizers, Appl. Soft Comput. 75 (2019) 775–789.
[24] R. Zamani, An evolutionary implicit enumeration procedure for solving the resource-constrained project scheduling problem, Int. Trans. Oper. Res. 24 (2017) 1525–1547.
[25] D. Paraskevopoulos, C. Tarantilis, G. Ioannou, Solving project scheduling problems with resource constraints via an event list-based evolutionary algorithm, Expert Syst. Appl. 39 (2012) 3983–3994.
[26] C. Fang, L. Wang, An effective shuffled frog-leaping algorithm for resource-constrained project scheduling problem, Comput. Oper. Res. 39 (2012) 890–901.
[27] J. Alcaraz, C. Maroto, A robust genetic algorithm for resource allocation in project scheduling, Ann. Oper. Res. 102 (2001) 83–109.
[28] L. Tseng, S. Chen, A hybrid metaheuristic for the resource-constrained project scheduling problem, Eur. J. Oper. Res. 175 (2006) 707–721.
[29] X. Zheng, L. Wang, A multi-agent optimization algorithm for resource constrained project scheduling problem, Expert Syst. Appl. 42 (2015) 6039–6049.
[30] S. Droste, T. Jansen, I. Wegener, On the analysis of the (1+1) evolutionary algorithm, Theor. Comput. Sci. 276 (2002) 51–81.
[31] A. Santiago, B. Dorronsoro, H. Fraire, P. Ruiz, Micro-genetic algorithm with fuzzy selection of operators for multi-objective optimization: μFAME, Swarm Evol. Comput. 61 (2021) 100818.
[32] Z. Liu, Z. Hua, L. Yang, R. Deng, Search space decomposition for resource-constrained project scheduling, Autom. Constr. 134 (2022) 104040.
[33] C. Coello, G. Pulido, A micro-genetic algorithm for multiobjective optimization, in: International Conference on Evolutionary Multi-criterion Optimization, 2001, pp. 126–140.
[34] R. Madadi, C. Balaji, Optimization of the location of multiple discrete heat sources in a ventilated cavity using artificial neural networks and micro genetic algorithm, Int. J. Heat Mass Transf. 51 (2008) 2299–2312.
[35] F. Au, Y. Cheng, L. Tham, Z. Bai, Structural damage detection based on a micro-genetic algorithm using incomplete and noisy modal test data, J. Sound Vib. 259 (2003) 1081–1094.
[36] Y. Toklu, Application of genetic algorithms to construction scheduling with or without resource constraints, Can. J. Civ. Eng. 29 (2002) 421–429.
[37] H. Zoulfaghari, J. Nematian, N. Mahmoudi, M. Khodabandeh, A new genetic algorithm for the RCPSP in large scale, Int. J. Appl. Evol. Comput. 4 (2013) 29–40.
[38] A. Halim, I. Ismail, S. Das, Performance assessment of the metaheuristic optimization algorithms: an exhaustive review, Artif. Intell. Rev. (2020) 1–87.
[39] M. Vanhoucke, J. Coelho, A tool to test and validate algorithms for the resource-constrained project scheduling problem, Comput. Ind. Eng. 118 (2018) 251–265.
[40] Z. Hua, Z. Liu, Y. Yang, L. Yang, Improved genetic algorithm based on time windows decomposition for solving resource-constrained project scheduling problem, Autom. Constr. 142 (2022) 104503.
[41] M. Abdar, F. Pourpanah, S. Hussain, D. Rezazadegan, L. Liu, M. Ghavamzadeh, P. Fieguth, X. Cao, A. Khosravi, U. Acharya, A review of uncertainty quantification in deep learning: techniques, applications and challenges, Inf. Fusion 76 (2021) 243–297.
[42] S. Chakraborty, D. Paul, S. Das, t-Entropy: a new measure of uncertainty with some applications, ArXiv Preprint, arXiv:2105.00316, 2021.
[43] C. Tzelepis, V. Mezaris, I. Patras, Linear maximum margin classifier for learning from uncertain data, IEEE Trans. Pattern Anal. Mach. Intell. 40 (2017) 2948–2962.
[44] A. Alkeshuosh, M. Moghadam, I. Al Mansoori, M. Abdar, Using PSO algorithm for producing best rules in diagnosis of heart disease, in: Proc. of International Conference on Computer and Applications, 2017, pp. 306–311.
[45] M. Zomorodi-moghadam, M. Abdar, Z. Davarzani, X. Zhou, P. Pławiak, U. Acharya, Hybrid particle swarm optimization for rule discovery in the diagnosis of coronary artery disease, Expert Syst. 38 (1) (2021) e12485.
[46] F. Wang, H. Zhang, A. Zhou, A particle swarm optimization algorithm for mixed-variable optimization problems, Swarm Evol. Comput. 60 (2021) 100808.
[47] Y. Duan, C. Liu, S. Li, Battlefield target grouping by a hybridization of an improved whale optimization algorithm and affinity propagation, IEEE Access 9 (2021) 46448–46461.