Applied Intelligence (2023) 53:5887–5909
https://doi.org/10.1007/s10489-022-03599-w

An effective parallel evolutionary metaheuristic with its application to three optimization problems

Mehrdad Amirghasemi[1]

Accepted: 7 April 2022 / Published online: 5 July 2022

Abstract
This paper presents a parallel evolutionary metaheuristic which includes different threads aimed at balancing exploration versus exploitation. Exploring different areas of the search space independently, each thread also communicates with other threads, and exploits the search space by improving a common high quality solution. The presented metaheuristic has been applied to three famous and hard-to-solve optimization problems, namely the job shop scheduling, the permutation flowshop scheduling, and the quadratic assignment problems. The results of computational experiments indicate that it is effective, versatile and robust, competing with the-state-of-art procedures presented for these three problems. In effect, in terms of solution quality, and average required running time to reach a high quality solution, the procedure outperforms several state-of-the-art procedures on multiple benchmark instances.

Keywords Parallel processing · Metaheuristics · Evolutionary computation · Flowshop scheduling · Job shop scheduling · Quadratic assignment · Facility layout

1 Introduction

Nature-inspired and evolutionary models have been exploited for solving a variety of hard-to-solve computational problems. One of the central benefits of these models is the facilitation of using parallel processing, which is the simultaneous usage of more than one processor cores in executing different parts of the same program. The key point with designing parallel processing in evolutionary computation is twofold. On the one hand, a program needs to be divided so that separate cores, without interfering with each other, can work together. On the other hand, the cores should communicate the best results they produce so that each core can use the experience of the other cores in evolving solutions.

With respect to this twofold consideration, this paper presents an evolutionary metaheuristic, in which different semi-independent threads, each running on a separate CPU core, work separately and interact occasionally with one another to inform each other about the best overall solution obtained. It is this best overall solution that becomes the focus of the extra search and its vicinity is searched by all the threads. According to the classification made by [24], the proposed parallel strategy can be considered as a *co-operative multi-search*.

A thread operates in three separate layers including (i) a heuristic construction module to generate initial solutions, (ii) a genetic algorithm module to combine high quality solutions, and (iii) an enhancing module to further improve the solutions. The enhancing module not only improves the best solution obtained by the corresponding thread, but enhances the best overall solution obtained by all other threads as well. The presented procedure, called the Parallel 3-layer Hybrid (P3H), considers each thread as the combination of six synergetic components, namely (i) an initial solution construction method, (ii) a crossover operation, (iii) a mutation operation, (iv) a restrictedTabu component, (v) a large neighborhood scheme, and (vi) a perturb mechanism.

To explore the search space independently and effectively, a thread uses its components in a randomized manner. The benefit of such randomization is twofold. First, it circumvents the possible search redundancy which could occur because all the threads are aimed at improving the same high

✉ Mehrdad Amirghasemi
mehrdad@uow.edu.au

1 SMART Infrastructure Facility, Faculty of Engineering and Information Sciences, University of Wollongong, Wollongong, New South Wales, Australia

quality solution. Second, even for a single thread, the randomization prevents it from doing the same set of operations if a solution has been encountered more than once.

The key contribution of this study is to show the effectiveness of this metaheuristic through synergetic integration of these six modules for three famous and hard-to-solve optimization problems, namely the job shop scheduling, the permutation flowshop scheduling, and the quadratic assignment problems. The rest of this paper is organized as follows. Section 2 discusses the related work for each of the problems. Section 3 presents the formulation of these three problems. The stepwise description of the P3H is outlined in Section 4, and the results of computational experiments are in Section 5. In Section 5, also the setting of parameters for the P3H with respect to each of the three problems is discussed. Concluding remarks and some suggestions for future work are described in Section 6.

2 Related work

The three problems on which the P3H is applied are Permutation Flowshop Scheduling Problem (PFSP), Job shop Scheduling Problem (JSP), and Quadratic Assignment Problem (QAP). While theses problem formulations are explained in detail in the next section, the related work is outlined here. In effect, this section is divided into two subsections. The first subsection describes related work to the components of the P3H, and the second subsection presents the related work to parallel methods for the PFSP, JSP, and the QAP.

2.1 Related work to the components of the presented method

The most effective strategies for hard-to-solve combinatorial optimization problems are categorized into the three groups of (i) constructive methods, (ii) local search techniques, and (iii) population-based methods. Constructive methods build a solution by sequentially deciding the values of solution components, i.e. decision variables. Imitating the survival of living creatures is the key idea used in the design of a popular constructive method called Ant Colony Optimization (ACO) [27]. The Rollout algorithm [15] is another popular constructive method capable of producing high quality solutions.

Unlike construction methods, local searches take a complete solution to a problem and by checking its immediate neighbors, which are similar solutions with one or two minor difference, aim to find an improved solution [71]. Getting stuck in local optima is the main problem with local search techniques and has been depicted in Fig. 1.

The problem with local optimality has been tried to be addressed in different ways. Whereas in the Iterated Local Search (ILS), the starting solution of the local search is derived by a *perturbation* of the previous local optimum found [44], in the Variable Neighborhood Search (VNS) a set of neighborhoods of different orders are employed [35]. On the other hand, Tabu Search (TS) explicitly exploits short-term and long-term memory to guide the search [33], with short-term memory being used to keep track of recently visited solutions and long-term memory to monitor the search progress.

Genetic Algorithms [36], Particle Swarm Optimization [22], and scatter search [33] are prime examples of population-based methods. Population based methods are composed of five main components, namely (i) an encoding/decoding scheme that maps every solution (*phenotype*) to a chromosome(*genotype*), (ii) a *fitness function* that assigns a goodness to each individual, (iii) a *parent selection* strategy which determines which individuals are nominated as parents to produce offspring, (iv) a *survival selection* strategy in which a rule is defined for deciding which individuals will be survived to the next generation, and (v) *reproduction operators*, which specify the way two or more encodings are combined to produce an offspring encoding.

Towards making population-based methods more effective, "go with the winners" strategy [5], population based incremental Learning (PBIL) [14], and path relinking and scatter search [34] concepts are instrumental. In effect, path relinking has been successfully applied to the QAP [2] and the PFSP [53].

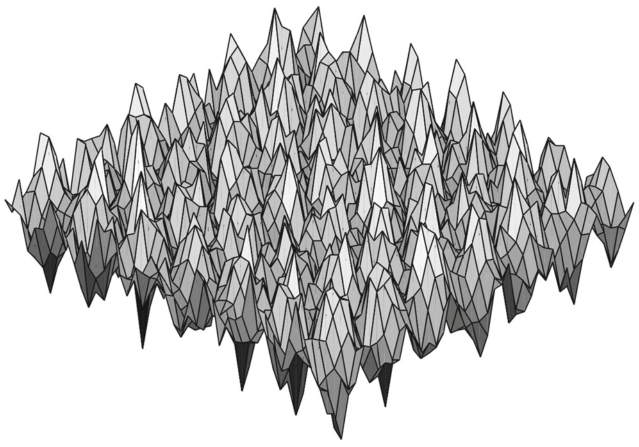

Fig. 1 A situation in which a local optimal solution has been surrounded by many other local optimal solutions

2.2 Parallel methods for the PFSP, JSP, and QAP

Since the early development of parallel processing technology in mainframes, many researchers have concentrated on solving hard-to-solve problems through integrating parallelization techniques with metaheuristics. In [23], a recent overview of parallel metahueristics was presented, and the promising performance of parallel *cooperative* strategies has been emphasized. In cooperative strategies, semi-independent procedures occasionally synchronize by sharing some information during the search progress. These methods are typically referred to as *cooperative multi-search* in the literature, and their success in tackling hard, NP-complete optimisation problems are further stressed in [4, 65, 67], and [24]. More recently, new techniques on adopting parallelization using Graphical Processing Units (GPUs) for well known metahuristics such as ACO and GA has been proposed [20, 50].

Single Program Multiple Data (SPMD) and Threads models could be seen as two commonly used parallel programming models employed in metaheursitcs. SPMD is a high-level parallel programming model in which all independent tasks run their copy of the same program simultaneously, with different input data or initial points. In the case of metaheuristics, this initial starting point could simply be a different seed value for the pseudorandom number generators. The threads model, on the other hand, is a type of shared memory programming, in which a single process can have several concurrent execution paths. Independent threads can also communicate with one another through a global (shared) memory. This is in line with parallel cooperative strategies, as the knowledge of search space can be shared and utilized by all threads.

It should be noted that multithreading is not necessarily equivalent to parallel computing. In this study, however, a threads model is used, and implemented using OpenMP API [21]. Provided that the number of threads is less than or equal to the number of CPU cores, OpenMP API typically ensures that each thread is run on a separate CPU core [21].

From the early beginning of parallel processing technology, interested researchers have focused on solving the PFSP, JSP, and QAP, as three highly-applicable and challenging problems through multithreaded and multi-core-based procedures. In the following subsections, these parallel procedures are surveyed for each of these problems, separately.

2.2.1 Parallel methods for the PFSP

As a source of parallelism, islands models in evolutionary searches can exchange individuals through the entire run of the algorithm, with this exchange of individuals being generally termed as "migration" [25, 73, 74]. Among several island model GAs proposed for the PFSP, we can mention the one presented in [59], where the authors conducted experiments on randomly generated instances with 40–100 jobs and 4–10 machines. Another island model is in [17], and has been presented for the FSP with total completion time criterion. In this island model, crossover operator is performed on individuals from different islands. The authors conducted experiments on Taillard's benchmarks [63].

The parallel simulated annealing of [75] is another related work. The authors suggest an island model parallel strategy in which cooperation occurs when the global best solution is being updated. The authors compared the independent and cooperative variant of the proposed procedure with NEH heuristic of [46] and concluded that the cooperative variant with four processors yields better results.

A parallel Tabu search has also been presented by the same authors [16] where the search threads cooperate by broadcasting the global best solution, wherein a specific thread is responsible for managing the exchange of global best solutions. The authors experimented on a selection of Taillard's benchmark instances [63] as well as some randomly generated instances.

Among the more recent parallel approaches are that of [70] and [52]. In [70] a cooperative island model has been proposed wherein, similar to our approach, same algorithm is run on different islands and occasional cooperation occurs at different stages in the algorithm. In the parallel hybrid proposed by [52], different allocations of Memetic Algorithms and the Iterated Greedy (IG) procedure [56], to multiple threads, have been studied.

2.2.2 Parallel methods for the JSP

Among the earliest parallel strategies, we can mention the parallel tabu search presented in [64], where the author presents a tabu search method and describes why it is more efficient than the shifting-bottle-neck procedure. In the same paper, a parallel variant of the tabu search has also been presented that divides a problem into k sub-problems, each containing a sub-set of jobs. Each of these sub-problems is then solved independently and, in the final stage, the sub-problems are aggregated to form a solution to the original problem.

Aiex et al. [3] present a hybrid of GRASP and path-relinking which operates on a pool of elite solutions. In effect, GRASP is used to generate a pool of elite solutions and path-relinking is applied to (i) a solution produced by GRASP and (ii) an elite solution chosen from the pool. The path-relinking result is used to update the pool of elite solutions. Also, two similar parallel variants have been proposed by [3], namely collaborative and non-collaborative. While the non-collaborative version is a SPMD approach in which each thread executes a copy of the algorithm, in the collaborative version the pool of elite-solutions is shared among threads. The authors also describe why their collaborative scheme presents a better speed-up factor than their non-collaborative scheme.

Another related work is that of [58] in which a parallel Variable Neighborhood Search (VNS) is proposed for the JSP. The authors' proposed VNS consists of two main components, namely *shake* and *LocalSearch* procedures. Whereas shake procedure has the role of perturbing a given solution, the employed LocalSearch procedure improves the given solution based on SWAP or INSERTION neighborhoods. Therefore, the proposed VNS of [58] consists of three main steps in each iteration: (i) constructing an initial solution, x, (ii) performing the shake procedure on x, and storing the result as x' (iii) performing the LocalSearch on x'.

Sevkli and Aydin [58] compared four different parallelization of their proposed VNS algorithm. In the first scheme, a copy of VNS is run by all processors, starting with a single initial solution. After the completion of VNS by all processors, the best solution found among all processors acts as the initial solution for the next iteration. While the first scheme waits for all threads to complete their VNS before starting the next generation, in the second scheme the parallelization strategy is asynchronous and as soon as a processor finishes its VNS run, its next iteration will start with the incumbent overall best solution at that time. The two other parallel schemes proposed by the authors are decentralized in the sense that the search threads communicate through a network of processors and no central synchronization is occurred. Two network structures proposed are: *unidirectional-ring*, and *mesh*. The authors argued that the ring topology has the best performance over all other schemes.

2.2.3 Parallel methods for the QAP

One of the earliest and famous procedures for the QAP is the Robust Tabu Search (RTS) developed in [62]. In that paper, the author presents two parallelization schemes for the proposed RTS: In the first scheme, the process of evaluating all potential neighbor solutions is divided among different processors for the purpose of reducing the computation time. In effect, after each processor evaluates its portion of neighborhood, all threads synchronize at a single point to identify the best possible neighbor. The other proposed parallel strategy of [62] is the SPMD approach of running RTS instances independently from different initial solutions.

Another related parallel procedure to our proposed algorithm is the Cooperative Parallel Tabu Search (CPTS) of [38], which incorporates the RTS of [62]. The CPTS is essentially an SPMD parallel algorithm in which a copy of RTS is run in each thread, and each thread stops and synchronizes before and after running the RTS procedure. In particular, the cooperation between processors occurs by maintaining a small set of elite solutions, named *reference set*. Basically, the number of solutions in the reference set is equal to the number of processors and the reference set is shared among all processors.

Among the other parallel strategies for the QAP, we can mention the parallel hybrid of ant-colony and tabu search [66], and the island model parallel genetic algorithm of [69]. In both methods, a master-slave paradigm is used and the global information regarding best solutions in islands and/or pheromone trail matrix is communicated among the processors.

The other related work is the single instruction multiple data tabu search (SIMD-TS) of [79], in which tabu search procedures are run on each thread, and probabilistically, certain diversification and intensification actions are performed in each thread. It is worth noting that the SIMD-TS has specifically been designed to be run on GPUs instead of ordinary CPUs. An extensive review of recent exact and heuristic methods for the QAP can be found in [29]

3 Problem formulation

The PFSP, JSP, and QAP can be successively described as follows. Since the PFSP is a special case of the Flowshop Scheduling Problem (FSP), the FSP needs to be discussed first.

The FSP is a subclass of scheduling problems in which n Jobs have to be processed on m machines, with the goal of finding an optimal processing sequence of jobs on machines. The optimality criterion is mainly the completion time of the last operation on the last machine (makespan). In the FSP, each job should be processed on the same sequence of machines. For instance, if job 1 should be done on machine 4, 2, 3, and 1, one after another, then all other jobs have the same order. The PFSP is a special case of the FSP in the sense that all machines have to process all the jobs in the same order. In the PFSP, assuming that the jobs are numbered $1, 2, \ldots, n$, the goal is to find an optimal permutation of jobs $\pi_1, \pi_2, \ldots, \pi_n$ so that the completion time of the last job on the last machine is minimized. The completion time of job π_i on machine j, $C(\pi_i, j)$, can be calculated as follows.

$$C(\pi_i, j) = \max\{C(\pi_{i-1}, j), C(\pi_i, j-1)\} + T(\pi_i, j) \quad (1)$$

where $C(\pi_0, j) = C(\pi_i, 0) = 0$ and $T(i, j)$ is the processing time of job i on machine j.

The PFSP has diverse applications in manufacturing and has increasingly attracted the attention of researchers to assess new algorithmic ideas. The PFSP is NP-hard [54, 55] and even some instances with a moderate number of jobs and machines have not been solved to optimality.

The second problem is the JSP. In the JSP, m jobs have to be processed on n machines and, unlike in the FSP, each job may have different processing order on machines. However, the same as in the FSP, each machine can process only one job at a time and each job can be processed only on one machine at a time. The goal is to find a schedule which minimizes the time required to process all jobs on all machines, i.e. the *makespan*. The JSP is one of the hardest combinatorial optimization problems [32]. An standard, well-known mathematical formulation for the JSP is disjunctive graph formulation, which can be found in [1].

The QAP is in the class of Facility-Location problems, with diverse applications in different areas such as factory layout design as well as the problem of placing electronic components in a circuit or microchip. In the QAP, we are given a set of locations and a set of facilities, both having the same size, n. In addition, two $n \times n$ matrices, D and F are given as input, with d_{kl} indicating pair-wise distance between locations k and l, and f_{ij} specifying pair-wise flow between facilities i and j. The objective is to find a one-to-one mapping between facilities and locations which minimizes the cost of flow, C, calculated as a summation of pair-wise flow between facilities multiplied by their distance.

A solution to the QAP can be simply represented as a permutation of facilities. For example, the permutation $\pi = (4, 3, 1, 2)$ represents the solution in which facilities 4, 3, 1, and 2, are placed in locations 1, 2, 3, and 4, respectively. Belonging to the class of NP-hard problems, the QAP is computationally demanding even with respect to finding a solution with the guarantee of being in a given distance of the optimal solution. With π representing the a solution and Π denoting the set of all possible permutations, the QAP objective function formulation is as follows.

$$\min_{\pi \in \Pi} C(\pi) = \sum_{i=1}^{n} \sum_{i=1}^{n} f_{ij} d_{\pi(i)\pi(j)} \quad (2)$$

It is worth noting that P3H could be applied as a general purpose metaheuristics, and any fitness or cost function and set of constraints could be adopted. Furthermore, any constraint in the given problem, could be seen as defining the feasible search space. In this paper, we identify a good solution, as the one having lower cost, i.e. a minimization problem, and the constraint are satisfied by considering only valid permutations, as feasible solutions.

4 The P3H

The P3H employs multi-threading, and in each thread a construction technique improves the quality of genomes in a genetic algorithm, with effective exploration/exploitation balance being achieved through an unrestricted and egalitarian parent selection. Also, restricted and elitist offspring selection contributes to the aforementioned balance. The threads are not fully independent in the sense that in each thread, the employed local search improves the best overall solution obtained in all threads. The three interacting layers of the P3H are depicted in Fig. 2, and the detailed stepwise description of the P3H has been shown in Fig. 3.

It should be noted that P3H can be simply reproduced for similar optimisation problems by (i) implementing the general, problem-independent modules, namely heuristic construction, genetic algorithm, and enhancing module, and (ii) incorporating six problem-specific modules. While the general modules are described next, the problem-specific components, outlined in Table 1, are explained in detail in the following subsections.

As is seen in Fig. 3, line 2 starts the threads, with the block of code instantiated for each thread being shown in lines 3 through 50. It is worth noting that since the P3H is a SPMD parallel approach, it is an identical copy of the same routine which is run by each thread.

Each thread operates in three layers. In the first layer, a number of high-quality solutions are constructed in the

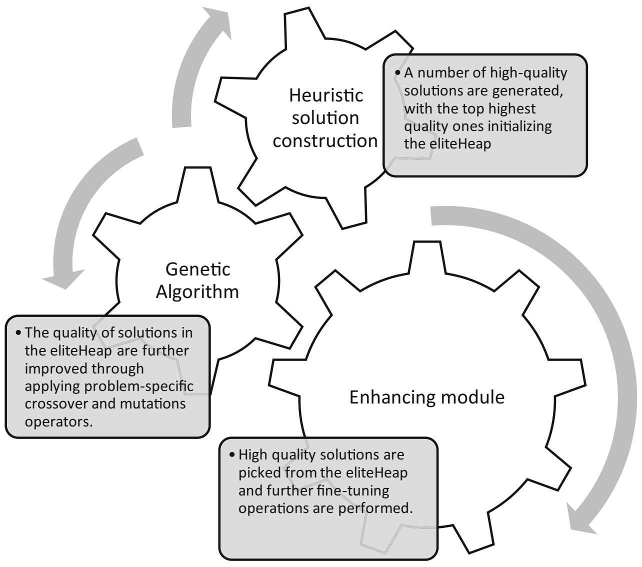
Heuristic solution construction
• A number of high-quality solutions are generated, with the top highest quality ones initializing the eliteHeap
Genetic Algorithm
• The quality of solutions in the eliteHeap are further improved through applying problem-specific crossover and mutations operators.
Enhancing module
• High quality solutions are picked from the eliteHeap and further fine-tuning operations are performed.

Fig. 2 Three layers of the P3H procedure and their interactions

Fig. 3 The pseudocode of parallel three-layer hybrid (P3H)

```
Procedure Parallel3LayerHybrid()
{
   #in each thread t: 1 ... T
   {
    //−−−−−−−−−−−−−−Layer 1: Heuristic construction module−−−−−−−−−−−−−−
    Construct $M$ solutions and put the top $m$ solutions in eliteHeap[t];
    //−−−−−−−−−−−−−−Layer 2: Genetic algorithm module−−−−−−−−−−−−−−−−−
    Construct an initial population[t] with size populationSize, which is less than $m$,
         from top $m$ solutions in layer 1;
    for (g=0 ; g<generationSize ; g++)
    {
     for (k=0 ; k<(populationSize/2) ; k++)
      {
       Select two members of the population[t] randomly and call them par1 and par2;
        if (the similarity between par1 and par2 is greater than similarityThreshold)
          replace the parent with lower quality with a solution from eliteHeap[t];
        Generate offsp1 and offsp2 based on crossover operations on par1 and
          par2;
        mutate offsp1 and offsp2, each with the chance of mutationProb;
      }//for (k=0 ...
      Update the current population with the highest quality solutions among
         parent and offspring results ;
    }//for (g=0 ...
    //−−−−−−−−−−−−−−Layer 3: Enhancing module−−−−−−−−−−−−−−
    Remove the highest quality solution from the final population[t] and call it X;
    while (The termination criterion is not met)
    {
       X=restrictedTabu(X);
       rnd=Random[0,1];
       switch (rnd)
       {
        case (rnd < intensificationProb)
          Y=Large-Neighborhood-Improvement(X)
          if (cost(Y)<cost(X))
            X=Y;
          else
            X=Perturb(X);
          break;
        case (rnd <concentrationProb+intensificationProb)
          Set X as the best global solution among all threads;
          X=Perturb(X)
          break;
        case (rnd<perturbationProb+concentrationProb+intensificationProb)
          X=Perturb(X);
          break;
        default // it is done with the chance RefreshProb
          Replace X with the next highest quality solution from previous levels;
          break;
       }//switch
      }//while
  }//#in each thread t
}//procedure
```

sense that among all of the constructed solutions, solutions with lower quality are ignored and those with higher quality are kept in a list called *eliteHeap*. Part of these kept solutions which have the highest quality are used in the second layer, and the rest can be used in the third layer.

In the second layer, an initial population of solutions is formed by selecting top solutions among the *eliteHeap*, with this population being evolved through the genetic algorithm. For the purpose of evolving the solutions, the algorithm works as follows. First, from the current population, whose size is shown with *populationSize*, randomly *populationSize*/2 pairs of parents are selected. Then, a problem-specific crossover operator is applied to each pair of parent genomes. This is followed by a mutation operator, so that an offspring population with the same size of the current population can be generated. Finally, from the

Table 1 The selection of P3H problem-specific modules for the PFSP, JSP, and QAP

Module	Adopted procedure		
	PFSP	JSP	QAP
solution construction	Re-blocking [9]	Randomized Giffler and Thompson [8]	GREEDY [11]
crossover	Longest Common Sub-sequence [37]	Linear Order crossover (LOX) [30]	Cohesive merge procedure [28]
mutation	Single swap and insertion	Single swap move on a random machine	Simple locations swap and random cyclic swaps
restrictedTabu	Modified tabu search [47]	Modified Limited Tabu Search (LTS) [8]	Extended N* [77]
large neighborhood	Iterative decomposition procedure [6]	Forward-Backward Shifting Bottleneck Procedure [7]	Neighborhood Decomposition-based Search [11]
perturbation	3-replacement move, and random insertion	Random Swap based on N1 [72] neighbourhood	Applying random cycles of size 3

combined population of parent and offspring genomes, only a half with the highest quality enter the next generation and the other half are ignored.

In the third layer, a fine-tuning procedure, which works based on a point-based strategy, is run on the solutions in the constructed pool, using the solutions left in the *eliteHeap*. As line 27 of the pseudo-code presented in Fig. 3 shows, this point-based strategy is a problem-specific restricted tabu search, called *restrictedTabu()*. It is restricted in the sense that after a maximum number of iterations with no improvement, the procedure is halted and the best solution encountered is returned. As mentioned, this restricted tabu strategy is problem-specific and is described for each problem in the next subsections.

Towards its fine-tuning process, the restricted tabu procedure is aimed at performing a number of actions for keeping a balance between intensification and diversification. As is indicated in the switch-case starting at line 29 of the pseudo-code in Fig. 3, this procedure randomly selects one of four different actions. The selection is made based on given probabilities as input parameters, and these four actions, from which stochastically an action in the switch-case is serially selected and performed, are as follows.

The first action is selected with the chance of *intensificationProb* and causes a problem-specific large neighborhood search to be performed on the current solution. Also, whenever this large neighborhood search is unable to improve the current solution, a perturbation is performed on the current solution, and replaces it with one of its neighbors.

The second action is selected with the chance of *concentrationProb*, and replaces the current solution with the global best solution among all threads. Then, in order to prevent the algorithm from revisiting the same solution, this global best solution undergoes a perturbation. The third action is chosen with the chance of *perturbationProb* and only causes a perturbation to be applied to the current solution. The main purpose of applying a perturbation is to diversify the search and assist the algorithm to escape local optima.

The fourth action is selected with the chance of *refreshProb*. Based on this action, the current solution is replaced with the unused highest quality solution from the previous layers of the same thread. The selection of a solution to replace the current solution is accomplished in the following order. Initially, the best unused solution of the final population constructed in second layer is considered and removed from the pool for providing a new solution. If there is no element left, however, the next highest quality solution left in the *eliteHeap* of the first layer is considered and removed from the *eliteHeap* for providing a new solution. In the case where both the pool and the *eliteHeap* have become empty, a new solution is constructed with the problem-specific construction method specified in the first layer.

It is worth noting that the sum of the four parameters of *intensificationProb*, *concentrationProb*, *perturbationProb*, and *refreshProb* is 1, implying that depending on different setting of these parameters, different balancing between intensification and diversification can be expected. Moreover, because of their constant sum, introducing only three parameters is enough and the remaining parameter can be calculated based on the given parameters.

As is shown in the pseudo-code of the P3H, the underlined modules are problem-specific and should be separately implemented for any specific problem. These problem-specific modules comprise the major components of the P3H and are as follows (i) an initial solution construction method, (ii) a crossover operation, (iii) a mutation operation, (iv) a restrictedTabu component, (v) a large neighborhood scheme, and (vi) a perturb mechanism.

The selection of these problem-specific modules is outlined in Table 1, and is described in detail for the PFSP, JSP, and QAP, respectively, in the three following subsections.

4.1 Problem-specific modules for the PFSP

The aforementioned six components, for the PFSP, have been implemented as follows. As the first component, the solution construction module, four different initial solution construction methods have been considered. These methods include (i) a uniformly-at-random construction method, in which all $n!$ job permutations have an equal chance of $(\frac{1}{n!})$ for being selected, (ii & iii), two re-blocking construction methods proposed in [9], and (iv) the RJP (Recursive Johnson Procedure) proposed in [6]. All of these methods have been tested in Section 5 and, based on the results of experiments, the re-blocking mechanism has been adopted. We denote these four methods with (i) UNIFORM, (ii) REBLOCK_I, (iii) REBLOCK_II, and (iv) RJP, respectively.

The other two components are the crossover and mutation operator. The Longest Common Sub-sequence (LCS) developed in [37] has been used as the crossover operator, and a single swap and insertion move, each with a chance of 0.5, have been used as the mutation operator. It is worth noting that the mutation operation is applied with a small probability, (0.1–0.2), given as an input parameter.

The choice of *restrictedTabu* module, as the next component, is based on a modified version of the tabu search proposed in [47]. In this modified implementation, first, an ordinary insertion local search is run on the current solution. Based on using the critical path information, this insertion local search avoids unfruitful moves. Moreover, by using the forward and backward completion time matrices, it reduces computation time [6, 9]. The *restrictedTabu* module stops when, for a fixed number of iterations, no improving move has been found.

The last two components are the Large-Neighborhood-Improvement and the Perturb module. The iterative decomposition procedure of [6] is used as the large neighborhood improvement. The method divides the solution into substrings and iteratively optimizes each substring, called *chunks*. The perturb module employed is the combination of a 3-replacement move, and a random insertion move. The 3-replacement move selects three random indexes, i_1, i_2, i_3, and puts the job at position i_1 in position i_2, the job at position i_2 in position i_3 and finally the job at position i_3 in position i_1. A parameter called *perturbIntensity* controls the degree of such perturbation by the number of times this 3-replacement move occurs. For instance, when this parameter is set to two, this 3-replacement move occurs two times.

4.2 Problem-specific modules for the JSP

The six components used for the JSP are as follows. With respect to the first component, solution construction, three initial methods have been tested. The First method is the forward-backward Semi-Active Schedule Generator (SASG) proposed in [7], and the second method is the forward-backward Randomized Giffler and Thompson (RGT) construction method proposed in [8]. This method primarily generates non-delay schedules using the well-known Giffler and Thompson procedure and probabilistically improves a non-delay schedule based on a forward-backward mechanism.

The other employed construction method, called Iterative Carlier (IC) procedure, uses the Carlier one-machine optimization procedure [19] and applies it iteratively. The process starts with an empty schedule, and, then, based on a random order of machines, it schedules the machines incrementally. In each iteration, through solving the associated one-machine problem, each single machine is scheduled to optimality, and the result updates the current solution. As will be discussed in Section 5, based on the computational experiments, among them only the RGT, has been adopted.

With respect to the second and third components, crossover and mutation operators, Linear Order crossover (LOX) of [30] and a simple mutation operator which performs a single swap move on a random machine have been considered.

Since the encoding used in the JSP is based on the simple permutation of operations on machines, LOX has to be individually applied to each pair of permutations on the same machine independently. However, the main problem with such independent application of LOX is that the resulting offspring solution may be infeasible. To rectify this possible infeasibility, we have used the Giffler and Thompson procedure to not only remove any infeasibility but create an active schedule as well.

The fourth component, the *restrictedTabu* module, has been implemented based on a modified version of the Limited Tabu Search (LTS) proposed in [8]. Whereas the same as the LTS, the employed module works based on N5 and N6’ neighborhoods [48], unlike in the LTS, the *restrictedTabu* module keeps no history of solutions visited during the search process. Furthermore, similar to LTS, for evaluating potential neighbor solutions, it uses an estimate of makespan and avoids any exact evaluation.

The fifth component, the Large-Neighborhood-Improvement module, has been adopted as the Forward-Backward Shifting Bottleneck Procedure (ForwardBackwardSBP) which has originally been proposed in [7]. The ForwardBackwardSBP works as a solution-refinement routine which incorporates the post-optimization stage of Shifting Bottleneck Procedure (SBP) of [1].

The Perturb module, as the sixth component, operates based on the N1 neighbourhood [72] and simply performs a random swap of two neighbouring operations on a random critical block. For this module, the *perturbIntensity* parameter determines the number of times a move is performed. The updating of the makespan and critical path, which need to be done after the completion of each move, are performed by the method presented in [8].

4.3 Problem-specific modules for the QAP

For the QAP, the six major components of the P3H have been set as follows. For the first component, the construction module, three construction methods have been implemented. The first method, named as UNIFORM, simply assigns facilities to locations uniformly at random. The next tested construction module has been developed in [77], and is called Randomized Heuristic (RH), aiming at assigning facilities having high interactions (flows) to locations having low distance to others. In this way, a percentage of facilities having high interactions are randomly selected and assigned to locations having small distances from one another. Then the remaining facilities, one after another, are allocated to empty locations in such a way the total assignment cost experiences the least increase. In effect, the second part of this module works similar to the procedure presented in [43].

The next construction method tested has originally been presented in [11] and is a modified implementation of the procedure presented in [43]. In the employed procedure, named as GREEDY, a solution is incrementally constructed by deciding for the best value of each solution component, one at a time, in a greedy manner. First a random permutation of facilities is generated, and, based on this order, the best location for each facility is determined and is added to the partial solution. In the Section 5, all these three constructed methods have been tested and, because of its higher performance, the GREEDY method has been adopted.

As the second component, the crossover operator, the cohesive merge procedure of [28] has been implemented. The cohesive merge is a computationally intensive crossover scheme which works based on the median distance of locations (sites). In our implementation, first, as an initialization phase, the median distance for all location is calculated. Next, given two parent solutions, every location is tested as a potential pivot site, resulting in $2n$ different solutions. From these $2n$ generated solutions, only the top two are selected as the offspring.

As the third component, the mutation operator, a combination of two different methods has been considered and applied randomly. The first operator is a simple swap of locations for two facilities, and the second method applies a random cycle of size 3. For instance, a cycle (f_1, f_2, f_3) indicates that facility f_1 should be re-assigned to the location of facility f_2, facility f_2 to the location of facility f_3, and facility f_3 in the empty location which had previously been occupied by facility f_1.

As the fourth component, the restrictedTabu module, the Extended N* local search procedure [77] has been adopted. This procedure starts with simple swap neighborhood, and continues using this neighborhood while improvement is possible. Then, it switches to N_λ neighborhood [43] in which after applying each degrading move, the two facilities relating to that swap are marked as Tabu for being prevented from participating in further iterations.

The fifth component, Large-Neighborhood-Improvement, has been selected from the Neighborhood Decomposition-based Search (NDS) developed in [11]. In this method, the relocation matrix associated with each solution of the QAP is passed to the Quick Match (QM) Linear Assignment solving routine [42], and the linear assignment proposals are extensively evaluated for improvement.

The sixth component, the Perturb module, simply applies a number of random cycles of size 3. The number of cycles applied is determined by the *perturbIntensity* parameter. For instance, when *perturbIntensity* is set to 5, the number of times the cycle is applied is five.

5 Computational experiments

The P3H has been coded in C++ and computational experiments have been performed on a Quad-Core PC with 3.4 GHz clock speed CPU. The Visual C++ has been used as the compiler and parallelization is based on OpenMP programming interface [21], with up to four cores of the CPU being used. Because of applying the P3H to three entirely different problems, a wide variety of benchmark instances have been involved in the experiments. For each of these three problems, their corresponding benchmark instances have been selected as follows.

For the PFSP, the Taillard's repository of benchmark instances [63], have been employed. Taillard's repository consists of 120 instances, comprising 12 classes, with 10 instances existing in each class. The number of jobs and machines of these instances are in the range 20–500 and 5–20, respectively.

For the JSP, 43 instances have been taken from the ORLIB website. This website is managed by Brunel University, UK, and contains a large number of benchmark instances produced for operational research problems.

Table 2 Comparing the performance of the three different construction methods for the PFSP

Instance	Size		UNIFORM		REBLOCK$_{\mathrm{I}}$		REBLOCK$_{\mathrm{II}}$		RJP	
	n	m	%DEV$_{\mathrm{avg}}$	%STDEV	%DEV$_{\mathrm{avg}}$	%STDEV	%DEV$_{\mathrm{avg}}$	%STDEV	%DEV$_{\mathrm{avg}}$	%STDEV
ta001	20	5	18.80	4.84	14.69	4.03	17.10	4.77	13.76	2.72
ta011	20	10	27.77	5.04	24.42	5.05	26.75	4.84	19.82	3.29
ta021	20	20	20.80	3.81	17.89	3.57	21.03	3.62	16.25	2.09
ta031	50	5	17.03	4.34	8.10	3.13	12.37	3.31	2.15	0.49
ta041	50	10	28.13	3.76	20.06	3.06	26.11	3.23	21.00	1.35
ta051	50	20	26.59	2.94	19.08	2.49	23.45	2.90	19.89	1.57
ta061	100	5	11.88	2.75	7.19	2.10	7.96	1.94	8.79	0.68
ta071	100	10	19.79	2.54	11.45	1.92	18.47	2.07	10.50	0.76
ta081	100	20	25.94	2.42	18.71	1.68	21.06	1.87	20.75	1.19
ta091	200	10	13.43	1.95	7.16	0.93	7.62	0.90	9.85	0.47
ta101	200	20	21.33	1.77	15.30	1.30	17.97	1.23	16.18	0.66
ta111	500	20	16.53	1.21	9.95	0.68	11.85	0.75	11.27	0.28
Average			20.67	3.11	14.50	2.49	17.64	2.62	14.19	1.30
Total time(s)			0.093		0.303		0.327		0.122	

These instances are (i) a selection of 11 hard-to-solve instances from [41], named *laxx*, (ii) 3 classic instances from [31] named ft06, ft10, and ft20, (iii) 4 instances yn1-4 from [76], (iv) 10 instances, swv01-10 from [60], (v) 5 instances abz5-9 due to [1], and (vi) 10 instances orb01-10 from [12].

For the QAP, a representative set of 29 benchmark instances, having up to 90 facilities/locations, have been selected from the QAPLIB [18]. These instances include 16 instances named *taixxa*, and *taixxb*, 7 instances named *skoxx* and a selection of 6 representative instances, namely els19, bur26d, nug30, ste36c, lipa50a, and tai64c. The number literals (*xx*) in the instance name indicate the instance size.

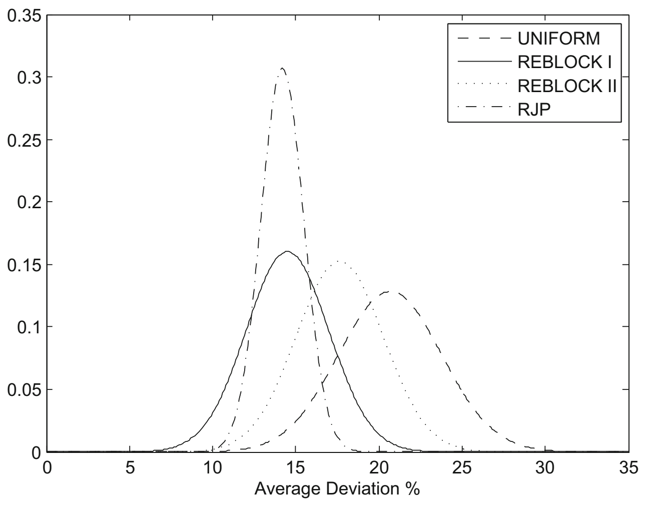

Fig. 4 The estimated normal distribution functions for the four different construction methods for the PFSP

It is worth mentioning that these instances are among the most popular instances in the literature.

Since the initial solution construction method plays a key role in affecting the overall quality of the solutions produced by the procedure, we first analyze the effect of selecting different initial solution construction methods for each of the three problems. Then, with respect to each problem, the values of parameters set for the P3H are discussed. Analyzing the parallelization effect is provided next. Finally, a brief evaluation is provided comparing the performance of the P3H with that of other state-of-the-art procedures.

5.1 Comparing the initial solution construction methods

As mentioned in Section 4.1, four heuristic construction methods have been implemented for the PFSP, namely (i) UNIFORM, (ii) REBLOCK$_{\mathrm{I}}$, (iii) REBLOCK$_{\mathrm{II}}$, and (iv) RJP, respectively. To evaluate these methods, a set of 12 Taillards' instances with different size has been used, and each construction method has been run 1000 times for each instance, with different random seed values. Table 2 compares these four methods. The column %DEV$_{\mathrm{avg}}$ shows the average percent deviation of solution Makespan (M) from the Upper Bound (UB), calculated as $(M-UB)/UB*100$. The column %STDEV shows the sample standard deviation percentage from the best known upper bound. In other words, $STDEV = \sqrt{\frac{1}{N-1} * \sum_{i=1}^{N} (M_i - M_{avg})^2}$ and $\%STDEV = STDEV/UB * 100$, with N being set to 1000, and M_i showing the makespan value for run i. Also,

Table 3 Comparing the performance of the three different construction methods for JSP

Instance	Size		SASG		RGT		IC	
	n	m	%DEV$_{avg}$	%STDEV	%DEV$_{avg}$	%STDEV	%DEV$_{avg}$	%STDEV
ft06	6	6	64.10	22.32	24.55	12.43	7.83	4.51
ft10	10	10	91.55	16.51	32.31	7.29	24.41	6.34
ft20	20	5	78.59	14.76	30.80	5.64	25.68	7.36
orb01	10	10	88.80	15.95	27.63	7.05	26.92	6.30
orb02	15	10	90.45	18.95	26.65	8.16	19.63	5.36
orb03	15	10	102.54	17.47	35.80	7.24	22.64	5.96
orb04	20	10	84.39	17.47	29.09	8.15	20.74	5.13
orb05	20	10	90.93	16.35	25.02	5.96	22.57	6.56
orb06	20	10	97.44	17.84	30.65	7.05	22.87	5.70
orb07	15	15	92.61	18.00	24.93	7.30	21.86	5.90
orb08	15	15	99.70	17.08	33.75	7.46	23.35	8.01
orb09	15	15	86.67	17.97	31.57	7.96	18.40	5.12
orb10	15	15	97.44	17.83	29.75	7.38	21.13	6.31
la19	15	15	91.35	20.66	19.23	6.94	16.93	5.03
la21	10	10	81.33	16.32	28.02	6.35	22.27	5.02
la24	10	10	79.71	16.31	29.93	8.12	21.53	5.46
la25	20	15	77.26	14.78	35.48	9.03	21.88	4.97
la27	20	15	79.15	13.82	27.92	5.81	27.59	5.67
la29	20	15	78.80	13.42	39.43	7.60	29.88	6.48
la36	10	10	74.16	13.98	32.79	7.81	22.16	4.79
la37	10	10	73.77	11.96	25.32	6.60	24.84	5.19
la38	10	10	86.46	14.35	31.73	8.30	26.70	5.42
la39	10	10	85.40	13.99	31.09	6.79	24.20	5.25
la40	10	10	85.04	13.68	28.14	7.47	25.02	5.23
abz5	10	10	82.08	17.66	18.36	7.37	12.74	3.98
abz6	10	10	83.41	19.16	17.85	6.80	10.40	4.32
abz7	10	10	71.32	10.24	28.60	5.94	29.01	5.21
abz8	10	10	83.01	12.08	33.88	6.44	29.72	5.01
abz9	10	10	81.97	10.91	32.94	5.75	34.37	5.33
yn1	20	20	76.77	10.36	28.39	6.34	27.03	4.11
yn2	20	20	71.37	10.29	29.83	6.08	28.02	4.47
yn3	20	20	79.13	10.74	26.03	5.89	29.94	4.49
yn4	20	20	85.48	10.75	26.29	5.08	32.73	5.08
swv01	20	10	107.09	12.57	41.20	6.31	39.91	7.65
swv02	20	10	89.85	13.62	36.90	6.12	32.13	6.61
swv03	20	10	98.82	13.00	38.74	5.94	37.08	7.82
swv04	20	10	106.02	13.17	37.01	5.45	36.19	7.16
swv05	20	10	101.06	13.60	41.78	6.30	33.66	6.53
swv06	20	15	103.25	12.69	41.33	5.72	39.48	6.57
swv07	20	15	98.86	11.71	39.93	6.03	34.12	5.36
swv08	20	15	114.57	12.27	39.68	5.45	42.28	6.90
swv09	20	15	109.63	12.21	43.98	6.32	39.51	6.74
swv10	20	15	105.05	12.69	36.05	5.45	35.40	6.03
Average			88.52	14.69	31.40	6.85	26.62	5.73
Total time(s)			2.403		1.401		157.635	

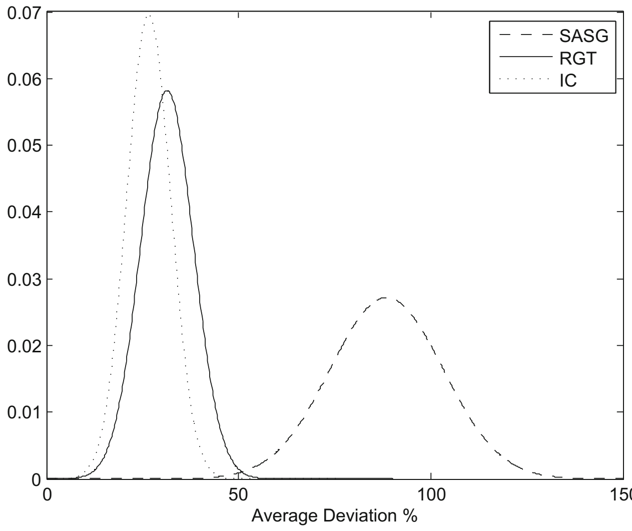

Fig. 5 The estimated normal distribution functions for the three different construction methods for the JSP

M_{avg} denotes the average makespan over 1000 runs and for each method, the total construction time for generating all 12000 solutions has also been shown in Table 2.

With the realistic assumption of having normal distribution for the $\%DEV_{avg}$, Fig. 4 shows the estimated probability density functions for all construction methods. The probability that a construction method generates solutions with zero percentage deviation from the best known solution (upper bound), i.e. $P(X \leq 0) = F_X(0)$, can be calculated for all four methods as $1.5 * 10^{-11}$, $2.8 * 10^{-9}$, $8.3 * 10^{-12}$, and $4.8 * 10^{-28}$, respectively. This indicates that the second method, the first re-blocking strategy, has the highest chance of generating solutions with zero percentage deviation from the best known solution.

In effect, the first method, as the closest competitor of the winner method, has to be at least 100 times faster than the winner method to outperform the winner, whereas comparing construction times shows that the first method is only 3 times faster than it. Hence, based on these observations, the first re-blocking method has been adopted as the construction module for the PFSP.

For selecting the best construction method for the JSP, all of the three presented methods have been considered, namely SASG, RGT, and IC. Each of these methods has been run with 1000 different random seed for each of the 43 JSP benchmark instances mentioned. As can be seen in Table 3, IC and RGT show superior performance over SASG. However, it should be noted that IC is considerably slower than both SASG and RGT. Figure 5 shows the estimated normal probability functions for the three methods. The probabilities $P(X \leq 0)$ for SASG, RGT, and IC can easily be calculated as $8.4 * 10^{-10}$, $2.3 * 10^{-6}$, and $1.7 * 10^{-6}$. Despite the fact that IC produces the highest quality solutions, its excessive required computational time prevents it from being selected. In effect, RC is more than 100 times slower than RGT and more than 60 times slower than SASG. Based on these considerations, RGT has been adopted as the construction module for the JSP.

For the QAP also, all of the three construction methods have also been compared and the best one has been adopted. The results have been shown in Table 4, and Fig. 6. In this table, $\%DEV_{avg}$ shows the average deviation of solution Cost (C) from the Best Known Solution (BKS), computed as $(C - BKS)/BKS * 100$. Similarly, %STDEV has been calculated as $\sqrt{\frac{1}{N-1} * \sum_{i=1}^{N} (C_i - C_{AVG})^2} * \frac{100}{BKS}$. The columns under the titles of UNIFORM, RH and GREEDY show the uniform, greedy, and RH methods, respectively, described in Section 4.3. As can be seen, the GREEDY method, despite being relatively slower, shows superior performance. Likewise, assuming normal distribution, the values of $P(X \leq 0)$ for UNIFORM, RH, and GREEDY are $1.6 * 10^{-9}$, $4.9 * 10^{-11}$, and $6.1 * 10^{-6}$, respectively. Therefore, based on these considerations, GREEDY has been adopted as the construction module for the QAP.

It should be noted that all the comparisons on initial solution construction methods for the PFSP, JSP and QAP, reported in this subsection, have been performed on the same running environment.

5.2 Parameter settings

Regardless of the problem the P3H applies to, the P3H needs totally 11 parameters for being adjusted. In setting these parameters, it has been noted that exploiting some high quality neighborhoods without effective exploration of solution space, and exploring solution space without effective exploitation of high quality neighborhoods is the easiest trap that an inappropriate parameter setting for the P3H can fall in. This theoretical principal is the main basis for manual setting of the P3H parameters. It is worth noting, while an adaptive (automatic) parameter tuning method [40] could be adopted, a manual parameter tuning, guided by balancing the exploration vs exploitation forces of the P3H is adopted and is explained as follows.

In effect, in setting these parameters for each of the three problems, a small number of instances, which with respect to their characteristics and sizes have had a varying degree of complexity, have been selected and different values for the parameters have been tested. Then the set of values which performed best have been chosen. In our parameter setting experiments, we found that many different settings of parameters achieved the same high quality performance.

This similar performance can be mainly explained by the robustness of the procedure. In setting the parameters, we

Table 4 Comparing the performance of the three different construction methods for the QAP

Instance	UNIFORM		RH		GREEDY	
	%DEV$_{avg}$	%STDEV	%DEV$_{avg}$	%STDEV	%DEV$_{avg}$	%STDEV
sko42	26.93	2.24	17.51	1.74	17.36	1.97
sko49	24.23	1.76	15.23	1.41	15.36	1.55
sko56	23.73	1.71	14.40	1.25	15.02	1.42
sko64	21.18	1.34	13.77	1.10	13.33	1.17
sko72	20.33	1.24	12.49	0.97	12.66	1.00
sko81	19.22	1.10	11.74	0.80	11.65	0.91
sko90	18.54	1.01	11.61	0.75	11.21	0.85
els19	241.64	61.13	163.96	42.84	206.63	56.10
bur26d	10.16	2.02	5.56	1.26	9.81	1.91
nug30	32.87	3.41	20.74	2.67	24.16	3.46
ste36c	127.66	20.88	73.59	11.60	99.53	20.43
lipa50a	3.12	0.16	2.57	0.14	2.65	0.14
tai64c	59.39	18.71	178.16	16.27	28.75	14.27
tai20b	163.04	50.76	99.15	28.12	104.69	41.05
tai25b	145.14	26.34	95.50	19.29	79.13	22.60
tai30b	105.26	17.52	83.58	15.43	51.59	13.65
tai35b	82.08	11.82	75.34	8.19	43.98	9.63
tai40b	77.50	8.86	61.14	6.38	40.17	7.04
tai50b	71.96	7.34	47.28	6.88	33.78	6.21
tai60b	65.86	6.24	39.14	2.82	34.69	5.56
tai80b	51.91	3.86	32.04	2.61	28.50	3.28
tai20a	27.52	3.38	19.50	2.89	17.85	2.66
tai25a	23.99	2.50	17.03	2.04	15.38	2.12
tai30a	20.89	1.80	15.23	1.60	13.53	1.51
tai35a	21.26	1.72	14.58	1.45	12.47	1.33
tai40a	20.51	1.53	14.04	1.24	12.18	1.27
tai50a	19.43	1.19	13.48	1.05	11.17	0.94
tai60a	18.21	1.01	12.53	0.82	9.92	0.74
tai80a	15.73	0.69	10.84	0.58	8.23	0.50
Average	53.77	9.08	41.09	6.35	33.98	7.77
Total time(s)	0.111		0.751		2.418	

noticed that it is based on the values of other parameters which the setting of a particular parameter finds its meaning. In effect, because of this interdependence, whereas in some contexts of the setting of other parameters, the lower values of a single parameter contribute to high quality solutions, in other contexts the reverse is true. This is partly due to the facts that (i) the performance of the P3H is mainly dependent on how it strikes balance between exploration and exploitation, and (ii) it is the combination of parameters that affects such balance.

In the cases where the increasing of a parameter, like *perturbationProb* or *mutation*, contributes to further exploration, and the increasing of the other parameter, like *intensificationProb* or *concentrationProb*, contributes to further exploitation, the setting of the first group of parameters can offset that of the second group and vice versa. Table 5 shows the setting of the parameters for each of the three problems, and it includes a UNIVERSAL column, proposing an initial starting value when applying P3H to other optimisation problems. This column is guided by our preliminary experiments as well as the average values for three problems under study. It is suggested that these universal starting values is adjusted based on both the exploration/exploitation considerations, explained in Section 4, and any relevant problem-specific knowledge.

5.3 Analyzing the effect of parallelization

Due to our Quad-Core processor and the capability of the P3H in working with different number of threads, we

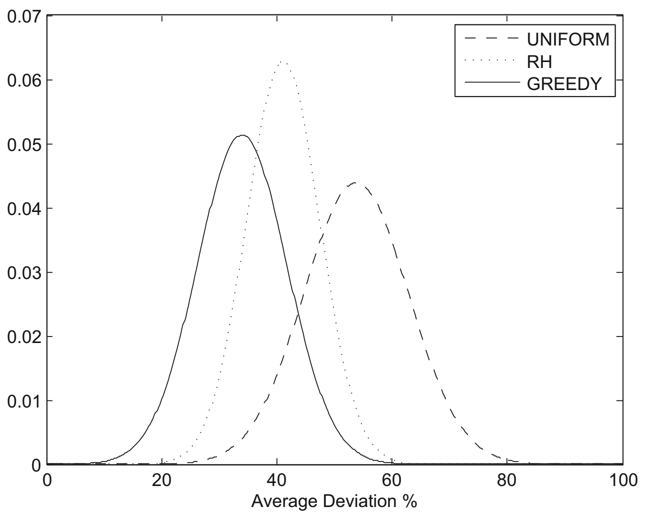

Fig. 6 The estimated normal distribution functions for the three different construction methods for the QAP

analyzed four variants of the P3H, shown as P3H$_c$, in which c shows the number of threads that are active. Although there is no limitation on the number of threads employed in the P3H, the Quad-Core processor has enforced the limit of four threads.

Each variant has been run 10 times with a problem-specific time limit for each run. The time limits have been set in such a way that the P3H has a maximum allowed running time equal or close to other algorithms in the literature. In this direction, for the PFSP and JSP, the time limit has been set to $n * m/10$ and max $\left(1.0, \frac{n}{m} * (9n - 60)\right)$ seconds for each run, where n and m are the number of jobs and machines, respectively. Also in solving the QAP, the time limit has been set to $\frac{1}{10} * 2^{\frac{N}{10}}$ minutes, where N shows the instance size. For each variant, the best and average percent deviation from the best known solution (tightest upper bound), %DEV$_{\text{best}}$ and %DEV$_{\text{avg}}$, has been reported. Additionally, to provide a basic estimate on the time complexity of the P3H$_c$, its maximum allowed run time, with respect to various sample problem sizes have been reported in Table 6.

Table 7 shows the result of running all four variants on the 12 classes of Taillards' instances. As can be seen, there is an improving trend with increasing the number of cores. For instance, we observe a 23% improvement in %DEV$_{\text{avg}}$ from 0.64 in the P3H$_1$ to 0.49 in the P3H$_4$.

As shown in Table 8, similar comparisons have been made for the JSP. For some of the JSP instances, increasing the number of cores from 2 to 3 has non-improving or degrading effect on %DEV$_{\text{avg}}$ and %DEV$_{\text{best}}$. This may be due to the unpredictable randomized nature of the metaheuristics in general and parallel metaheuristics in particular which can lead to unexpected behaviors. In effect, the issue is more critical for parallel methods due to parallel computational overhead and possible memory race conditions for the shared variables. It should be noted that despite the increase of %DEV$_{\text{avg}}$ for some individual instances, the total %DEV$_{\text{avg}}$ for the P3H$_2$ is equal to that of the P3H$_3$, and %DEV$_{\text{best}}$ is decreased from 0.34 to 0.30, indicating significant improvement for the rest of the instances.

For the QAP instances, Table 9 shows the related results. Again, an improving trend can be observed in both %DEV$_{\text{avg}}$ and %DEV$_{\text{best}}$. In effect, %DEV$_{\text{avg}}$ has been improved nearly by 25% through increasing the number of cores from 1 to 4.

Table 5 The setting of parameters for the PFSP, JSP, and QAP

Parameter	PFSP value	JSP value	QAP value	UNIVERSAL value
Layer 1:				
totalSolutionsGenerated	10000	2000	10000	10000
eliteHeapSize	1000	100	500	500
Layer 2:				
poolSize	100	50	100	100
generationSize	200	50	200	100
similarityThreshold	0.75	0.95	0.75	0.8
mutationProb	0.2	0.2	0.2	0.2
Layer 3:				
intensificationProb	0.2	0.1	0.4	0.2
concentrationProb	0.4	0.2	0.1	0.2
perturbationProb	0.1	0.1	0.1	0.1
perturbationIntensity	1	1	2	2
refreshProb	0.3	0.6	0.4	0.4

Table 6 Maximum allowed running times of the P3H, for the PFSP, JSP, and QAP, respectively

PFSP			JSP			QAP	
n	m	Time(s)	n	m	Time(s)	n	Time(m)
20	5	10	10	10	30	30	0.8
20	10	20	15	10	112.5	40	1.6
20	20	40	15	15	75	50	3.2
50	10	50	20	5	480	60	6.4
100	10	100	20	10	240	72	14.7
100	20	200	20	15	160	80	25.6
200	20	400	20	20	120	90	51.2

Finally, in order to compare different variants of P3H in the same running environment, three variants of the P3H have been implemented and are run on the same CPU, each using a single thread/core, having identical maximum running time. These variants are $P3H_{GA}$, $P3H_{ENHANCE}$, and $P3H_1$. While in $P3H_{GA}$ the enhancing module of the P3H is disabled and only the genetic algorithm layer is active, in $P3H_{ENHANCE}$, only the enhancing layer, is active, and the GA layer is deactivated. In $P3H_1$, while both GA and enhancing layers are active, the procedure operates on a single thread/core. The results are shown in Table 10. As can be seen, $P3H_{ENHANCE}$ and $P3H_1$ show a superior performance compared to $P3H_{GA}$. This highlights the crucial contribution of the enhancing module to the overall success of the P3H.

5.4 Comparison with other metaheuristics

In this section, we compare the $P3H_4$ with some of the highest performance metaheuristic in the literature. The comparisons are based on $\%DEV_{avg}$ and the running time. It is worth noting that since different programming approaches have been used and various computational environments and threads have been employed, the reported running times have only informative purposes and any running time comparison needs to consider the discrepancies involved, from the number of threads through the type of processors to the categories of the programming environments.

Table 11 shows the result of comparing the $P3H_4$ with several state-of-the-art metaheuristics. These methods include the Simulated Annealing algorithm, SAOP, of [49], the iterated local search (ILS) of [61], two Ant Colony Optimization algorithms, PACO and M-MMAS, proposed in [51], the Hybrid genetic algorithm, HGA_RMA, of [57], the Particle swarm optimization algorithm, PSO_{VNS} of [68], the hybrid variable neighborhood search, $NEGA_{VNS}$ of [80], the Re-blocking Adjustable Memetic Procedure, RAMP, of [9], and the Refining Decomposition-based Integrated Search, RDIS, of [6].

Table 7 Analyzing the effect of the P3H parallelization on $\%DEV_{best}$ and $\%DEV_{avg}$ for the PFSP

Instances	Size		$P3H_1$		$P3H_2$		$P3H_3$		$P3H_4$	
	n	m	$\%DEV_{best}$	$\%DEV_{avg}$	$\%DEV_{best}$	$\%DEV_{avg}$	$\%DEV_{best}$	$\%DEV_{avg}$	$\%DEV_{best}$	$\%DEV_{avg}$
ta001-ta010	20	5	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
ta011-ta020	20	10	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.00
ta021-ta030	20	20	0.00	0.01	0.00	0.01	0.00	0.01	0.00	0.01
ta031-ta040	50	5	0.01	0.06	0.01	0.03	0.00	0.02	0.00	0.01
ta041-ta050	50	10	0.39	0.60	0.34	0.47	0.31	0.48	0.30	0.43
ta051-ta060	50	20	0.49	0.81	0.43	0.71	0.35	0.59	0.31	0.58
ta061-ta070	100	5	0.02	0.11	0.01	0.06	0.00	0.05	0.00	0.04
ta071-ta080	100	10	0.23	0.50	0.16	0.37	0.13	0.33	0.05	0.30
ta081-ta090	100	20	1.23	1.61	1.03	1.41	1.01	1.32	0.99	1.30
ta091-ta100	200	10	0.26	0.48	0.18	0.34	0.15	0.33	0.16	0.30
ta101-ta110	200	20	1.68	2.07	1.52	1.88	1.43	1.73	1.36	1.69
ta111-ta120	500	20	1.14	1.46	1.09	1.37	1.03	1.28	1.01	1.22
Average			0.45	0.64	0.40	0.55	0.37	0.51	0.35	0.49

Table 8 Analyzing the effect of the P3H parallelization on %DEV$_{best}$ and %DEV$_{avg}$ for the JSP

Instance	Size		$P3H_1$		$P3H_2$		$P3H_3$		$P3H_4$	
	n	m	%DEV$_{best}$	%DEV$_{avg}$	%DEV$_{best}$	%DEV$_{avg}$	%DEV$_{best}$	%DEV$_{avg}$	%DEV$_{best}$	%DEV$_{avg}$
ft06	6	6	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
ft10	10	10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
ft20	20	5	0.00	0.21	0.00	0.14	0.00	0.00	0.00	0.00
la19	10	10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
la21	15	10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
la24	15	10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
la25	20	10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
la27	20	10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
la29	20	10	0.09	0.86	0.09	0.59	0.69	0.91	0.09	0.66
la36	15	15	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
la37	15	15	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.00
la38	15	15	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
la39	15	15	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
la40	15	15	0.00	0.15	0.16	0.16	0.00	0.15	0.00	0.15
abz5	10	10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
abz6	10	10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
abz7	20	15	0.15	0.58	0.15	0.29	0.15	0.34	0.15	0.29
abz8	20	15	0.30	0.59	0.45	0.71	0.60	0.71	0.30	0.56
abz9	20	15	0.00	0.69	0.00	0.28	0.00	0.69	0.00	0.37
orb01	10	10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
orb02	10	10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
orb03	10	10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
orb04	10	10	0.00	0.12	0.00	0.18	0.00	0.00	0.00	0.00
orb05	10	10	0.00	0.00	0.00	0.05	0.00	0.00	0.00	0.00
orb06	10	10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
orb07	10	10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
orb08	10	10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
orb09	10	10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
orb10	10	10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
yn1	20	20	0.23	0.71	0.00	0.51	0.00	0.44	0.00	0.44
yn2	20	20	0.33	0.73	0.33	0.73	0.33	0.60	0.00	0.49
yn3	20	20	0.34	0.65	0.34	0.53	0.22	0.63	0.22	0.49
yn4	20	20	0.52	1.32	0.31	0.98	0.31	0.91	0.21	0.79
swv01	20	10	0.92	2.22	1.21	2.03	0.36	2.00	0.36	1.56
swv02	20	10	0.54	1.00	0.34	0.82	0.34	0.80	0.34	1.09
swv03	20	10	1.57	2.42	1.50	2.09	1.57	2.25	1.22	2.05
swv04	20	10	1.56	2.80	1.63	2.80	1.29	2.71	1.16	2.44
swv05	20	10	1.12	2.19	0.42	1.52	0.56	1.52	0.28	1.53
swv06	20	15	1.26	2.45	1.08	1.92	1.50	2.18	0.42	1.28
swv07	20	15	2.07	2.95	1.25	2.56	1.38	2.71	1.57	2.58
swv08	20	15	1.20	3.06	1.77	2.72	0.80	2.53	1.60	2.23
swv09	20	15	1.15	2.89	1.69	2.84	1.09	2.76	1.39	2.58
swv10	20	15	2.24	3.20	1.89	2.89	1.66	2.67	1.55	2.58
Average			0.36	0.74	0.34	0.64	0.30	0.64	0.25	0.56

Table 9 Analyzing the effect of the P3H parallelization on $\%DEV_{best}$ and $\%DEV_{avg}$ for the QAP

Instance	$P3H_1$		$P3H_2$		$P3H_3$		$P3H_4$	
	$\%DEV_{best}$	$\%DEV_{avg}$	$\%DEV_{best}$	$\%DEV_{avg}$	$\%DEV_{best}$	$\%DEV_{avg}$	$\%DEV_{best}$	$\%DEV_{avg}$
sko42	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
sko49	0.00	0.03	0.00	0.01	0.00	0.01	0.00	0.02
sko56	0.01	0.05	0.00	0.02	0.00	0.02	0.00	0.02
sko64	0.00	0.03	0.00	0.02	0.00	0.01	0.00	0.00
sko72	0.00	0.05	0.00	0.04	0.00	0.04	0.00	0.03
sko81	0.01	0.06	0.01	0.04	0.01	0.05	0.01	0.04
sko90	0.00	0.05	0.02	0.05	0.03	0.06	0.00	0.02
els19	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
bur26d	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
nug30	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
ste36c	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
lipa50a	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
tai64c	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
tai20b	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
tai25b	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
tai30b	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
tai35b	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.00
tai40b	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
tai50b	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.00
tai60b	0.00	0.01	0.00	0.01	0.00	0.00	0.00	0.00
tai80b	0.00	0.04	0.00	0.03	0.00	0.03	0.00	0.01
tai20a	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
tai25a	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
tai30a	0.00	0.02	0.00	0.00	0.00	0.00	0.00	0.00
tai35a	0.00	0.19	0.00	0.19	0.00	0.15	0.00	0.10
tai40a	0.37	0.54	0.07	0.40	0.31	0.45	0.12	0.37
tai50a	0.47	0.84	0.68	0.86	0.68	0.82	0.39	0.61
tai60a	0.90	1.01	0.75	0.96	0.60	0.86	0.64	0.85
tai80a	0.92	1.11	0.78	1.03	0.83	1.05	0.82	0.97
Average	0.09	0.14	0.08	0.13	0.08	0.12	0.07	0.11

As can be seen in Table 11, the $P3H_4$ outperforms all other approaches on 20×20 and 50×20 problem groups. In addition, for 20×5, 20×10 and 100×20 instances, the $P3H_4$ performance is equal to that of top performing methods. Furthermore, except for three problem groups with the deviation percentages of 1.3%, 1.69%, and 1.22%, the average deviation from the best known solution is always less than 0.6%.

Concerning the processor speeds, it should be mentioned that HGA_RMA, $NEGA_{VNS}$, and PSO_{VNS} have been run on PCs with 2.6 GHz, 2.4 GHz, and 2.8 GHz clock speeds respectively. Also both RDIS and RAMP have been run on a 2.2 GHz CPU. Table 12 shows the maximum allowed running time, in seconds, for different approaches. As can be seen, $NEGA_{VNS}$, RDIS, RAMP, and the $P3H_4$ have been allowed an equal time limit and HGA_RMA has the lowest running time. It should be noted that SAOP, ILS, M-MMAS, and PACO have all been re-implemented in [57], and hence they are run on the same processor, and share the same maximum allowed running time, as that of HGA_RMA.

Further, since $NEGA_{VNS}$, RAMP, RDIS, and $P3H_4$ have identical maximum allowed running time, Friedman test, on their relative ranks, presented in Table 13, has been performed. An interested reader can see [26] for different tests applied to metaheuristics performance. It should be noted that ranks are calculated based on average deviation from the best known solution, $\%DEV_{avg}$. In the case of tied $\%DEV_{avg}$ values, the average of the ranks is used. As can be seen, whereas RAMP and $P3H_4$ have the lower ranks on smaller instances, $NEGA_{VNS}$ shows better performance

Table 10 Comparing the performance of the three different variants of the P3H

Instance	Size		$P3H_{GA}$		$P3H_{ENHANCE}$		$P3H_1$	
	n	m	$\%DEV_{best}$	$\%DEV_{avg}$	$\%DEV_{best}$	$\%DEV_{avg}$	$\%DEV_{best}$	$\%DEV_{avg}$
ta001	20	5	0.00	0.03	0.00	0.00	0.00	0.00
ta011	20	10	0.25	1.23	0.00	0.00	0.00	0.00
ta021	20	20	0.78	1.10	0.00	0.00	0.00	0.00
ta031	50	5	0.00	0.00	0.00	0.00	0.00	0.00
ta041	50	10	2.07	3.23	1.14	1.14	1.10	1.17
ta051	50	20	2.49	3.67	0.42	0.83	0.44	0.85
ta061	100	5	0.00	0.00	0.00	0.00	0.00	0.00
ta071	100	10	0.33	0.81	0.02	0.16	0.16	0.30
ta081	100	20	3.37	4.65	1.44	1.68	1.35	1.58
ta091	200	10	0.21	0.74	0.09	0.34	0.11	0.28
ta101	200	20	2.41	3.43	1.32	1.58	1.35	1.68
ta111	500	20	1.40	1.97	1.21	1.52	1.22	1.56
Average			1.11	1.74	0.47	0.60	0.48	0.62

on larger instance groups. Friedman test, performed on the ranked data set, shows a statistically significant difference of the ranks, with $\chi^2(2) = 15.286$ and $p = 0.002$.

The performance of the procedure on the JSP has also been compared with several metaheuristics as shown in Table 14. These methods are (i) Small-Large Embedded Neighborhood Search (SLENP) of [7] (ii) Tabu-based Genetic Algorithm (TGA) of [8], (iii) Tabu Search and Simulated Annealing (TSSA) hyrid of [78], and Iterative Ejections of Bottleneck Operations (IEBO) procedure of [45]. The column T_{avg} shows the average running time, in seconds, to find the best solution for each method. Both SLENP and TGA have been run on a PC with 2.2 GHz CPU, TSSA has been run on a 3.0 GHz CPU, and IEBO has been run on a cluster with 2.93 GHz CPU cores.

As shown in Table 14, in terms of deviation from the best known solution, the $P3H_4$ outperform both SLENP and TGA, with an overall $\%DEV_{avg}$ of 0.56%, compared to 1.11% and 1.28% deviations of SLENP and TGA, respectively. Moreover, on *orb* instances, $P3H_4$ outperforms SLENP, TGA, and TSSA with a $\%DEV_{avg}$ of 0.00% for all instances. IEBO has the lowest $\%DEV_{avg}$ on *swv* and *yn* instances, at the expense of significantly higher running times, compared to that of three other methods.

With respect to the QAP, the performance of the $P3H_4$ has been compared with that of the Progressive Adjusting

Table 11 Comparison of the $\%DEV_{avg}$ of $P3H_4$ to that of other metaheuristics for the PFSP

Instances	Size		SAOP	ILS	M-MMAS	PACO	HGA_RMA	PSO_{VNS}	$NEGA_{VNS}$	RAMP	RDIS	$P3H_4$
	n	m										
ta001-ta010	20	5	1.05	0.33	0.04	0.18	0.04	0.03	0.00	0.00	0.00	0.00
ta011-ta020	20	10	2.60	0.52	0.07	0.24	0.02	0.02	0.01	0.03	0.03	0.00
ta021-ta030	20	20	2.06	0.28	0.06	0.18	0.05	0.05	0.02	0.04	0.04	0.01
ta031-ta040	50	5	0.34	0.18	0.02	0.05	0.00	0.00	0.00	0.00	0.01	0.01
ta041-ta050	50	10	3.50	1.45	1.08	0.81	0.72	0.57	0.82	0.37	0.95	0.43
ta051-ta060	50	20	4.66	2.05	1.93	1.41	0.99	1.36	1.08	0.61	1.88	0.58
ta061-ta070	100	5	0.30	0.16	0.02	0.02	0.01	0.00	0.00	0.00	0.01	0.04
ta071-ta080	100	10	1.34	0.64	0.39	0.29	0.16	0.18	0.14	0.06	0.38	0.30
ta081-ta090	100	20	4.49	2.42	2.42	1.93	1.30	1.45	1.40	1.76	2.35	1.30
ta091-ta100	200	10	0.94	0.50	0.30	0.23	0.14	0.18	0.16	0.15	0.32	0.30
ta101-ta110	200	20	3.67	2.07	2.15	1.82	1.26	1.35	1.25	2.00	2.30	1.69
ta111-ta120	500	20	2.20	1.20	1.02	0.85	0.69	–	0.71	1.20	1.32	1.22

Table 12 Comparison of the maximum allowed running times of the P3H$_4$ to that of other metaheuristics for the PFSP

Instances	Size		SAOP	ILS	M-MMAS	PACO	HGA_RMA	PSO$_{VNS}$	NEGA$_{VNS}$	RAMP	RDIS	P3H$_4$
	n	m										
ta001-ta010	20	5	4.5	4.5	4.5	4.5	4.5	300	10	10	10	10
ta011-ta020	20	10	9	9	9	9	9	300	20	20	20	20
ta021-ta030	20	20	18	18	18	18	18	300	40	40	40	40
ta031-ta040	50	5	11.3	11.3	11.3	11.3	11.3	300	25	25	25	25
ta041-ta050	50	10	22.5	22.5	22.5	22.5	22.5	300	50	50	50	50
ta051-ta060	50	20	45	45	45	45	45	300	100	100	100	100
ta061-ta070	100	5	22.5	22.5	22.5	22.5	22.5	600	50	50	50	50
ta071-ta080	100	10	45	45	45	45	45	600	100	100	100	100
ta081-ta090	100	20	90	90	90	90	90	600	200	200	200	200
ta091-ta100	200	10	90	90	90	90	90	600	200	200	200	200
ta101-ta110	200	20	180	180	180	180	180	600	400	400	400	400
ta111-ta120	500	20	450	450	450	450	450	–	1000	1000	1000	1000

Structural Solver (PASS) of [10], the Neighborhood Decomposition-based Search (NDS) of [11], Self-Adaptive Facility Interchange (SAFI) of [77], the Diversified Tabu Search (DivTS) of [39], and the Cooperative Parallel Tabu Search(CPTS) of [38]. Since each study, except for CPTS, have reported performance only on a subset of instances under study, it is difficult to make any conclusion based on overall averages. However, it can be seen that P3H$_4$ outperforms PASS, NDS, SAFI, and ACO-GA/LS on *sko* instances. Overall, it seems the CPTS is the highest performing algorithm for the QAP at the time. The CPTS uses a parallel approach and has been run on 10 CPUs each having 1.3 GHz clock speed. In Table 15, the average running time (T_{avg}) of all algorithm, except for P3H$_4$, has been reported, in minutes. However, since the P3H$_4$ works based on time-limit stopping criterion, the maximum allowed time T_{max} has been reported. As is seen, while having a close overall performance to the CPTS, the P3H$_4$ is significantly faster than it.

6 Concluding remarks

With respect to solving three hard-to-solve combinatorial optimization problems, the P3H has been both robust and effective, achieving 0.35%, 0.25%, and 0.07% overall average best deviation (%DEV$_{best}$) for the PFSP, JSP and QAP, respectively. This is due to two complementary facts. First, the employed threads use components in random order and in this way explore solution space effectively. Second, the threads cooperate by improving a common solution and this way exploit high quality areas of solution space. It is this balance between exploration and exploitation which has significantly contributed to the robustness and efficiency of the P3H. It is worth noting that the success of the P3H in solving the flowshop scheduling problem could have multiple practical implications and benefits with respect to total energy consumption in production systems [13].

Directions for overcoming limitations and further enhancing the procedure are as follows. First, the parallel strategy employed can be enhanced by creating a pool of high quality solutions and providing access to the elements of this pool for all of the threads. In this way, the threads fill and cooperatively improve the same pool. Second, a feedback mechanism can be incorporated into the procedure to take the responsibility of further balancing exploration with exploitation. This balancing can be done by embedding a learning capability in the procedure to adjust the parameters while a problem is being solved, and can be of paramount importance.

Table 13 The ranks of NEGA$_{VNS}$, RAMP, RDIS, and P3H$_4$ based on average deviation from the best known solution

Instance group	NEGA$_{VNS}$	RAMP	RDIS	P3H$_4$
1	2.5	2.5	2.5	2.5
2	2	3.5	3.5	1
3	2	3.5	3.5	1
4	1.5	1.5	3.5	3.5
5	3	1	4	2
6	3	2	4	1
7	1.5	1.5	3	4
8	2	1	4	3
9	2	3	4	1
10	2	1	4	3
11	1	3	4	2
12	1	2	4	3

Table 14 Comparison of the %DEV$_{avg}$ and T$_{avg}$ of the P3H$_4$ to that of other metaheuristics for the JSP

Instance	Size		SLENP		TGA		TSSA		IEBO		P3H$_4$	
	n	m	%DEV$_{avg}$	T$_{avg}$	%DEV$_{avg}$	T$_{avg}$	%DEV$_{avg}$	T$_{avg}$	%DEV$_{avg}$	T$_{avg}$	%DEV$_{avg}$	T$_{avg}$
ft06	6	6	0.00	0.0	0.00	0.0	–	–	–	–	0.00	0.0
ft10	10	10	0.00	9.2	0.00	0.7	0.00	3.8	–	–	0.00	0.5
ft20	20	5	0.00	2.7	0.00	92.1	–	–	–	–	0.00	22.0
la19	10	10	0.00	0.8	0.00	1.2	0.00	0.5	–	–	0.00	0.2
la21	15	10	0.07	15.2	0.35	26.5	0.00	15.2	–	–	0.00	16.8
la24	15	10	0.16	20.4	0.03	51.2	0.13	19.8	–	–	0.00	1.6
la25	20	10	0.00	13.7	0.20	17.1	0.01	13.8	–	–	0.00	2.2
la27	20	10	0.00	32.0	0.08	45.3	0.00	11.7	–	–	0.00	5.1
la29	20	10	1.00	40.0	1.27	22.5	0.63	63.9	–	–	0.66	36.3
la36	15	15	0.02	12.9	0.00	5.1	0.00	9.9	–	–	0.00	5.5
la37	15	15	0.00	9.2	0.18	14.8	0.39	42.1	–	–	0.00	5.4
la38	15	15	0.22	14.8	0.14	5.7	0.30	47.8	–	–	0.00	6.0
la39	15	15	0.05	19.1	0.06	6.2	0.06	28.6	–	–	0.00	1.5
la40	15	15	0.38	15.9	0.29	11.1	0.20	52.1	–	–	0.15	8.7
abz5	10	10	0.00	3.5	0.05	11.5	–	–	–	–	0.00	1.1
abz6	10	10	0.00	0.2	0.00	0.1	–	–	–	–	0.00	0.1
abz7	20	15	1.22	64.8	1.36	50.7	0.88	85.9	–	–	0.29	60.4
abz8	20	15	1.11	55.9	1.44	38.2	0.80	90.7	–	–	0.56	33.3
abz9	20	15	1.70	35.8	1.28	33.9	0.85	90.2	–	–	0.37	50.3
orb01	10	10	0.06	6.0	0.17	5.6	0.00	3.5	–	–	0.00	0.9
orb02	10	10	0.00	0.5	0.00	0.7	0.01	6.4	–	–	0.00	0.2
orb03	10	10	0.00	7.4	0.57	1.4	0.75	13.8	–	–	0.00	0.4
orb04	10	10	0.12	7.6	0.56	4.8	0.33	14.3	–	–	0.00	5.1
orb05	10	10	0.00	12.1	0.00	10.0	0.18	6.6	–	–	0.00	5.8
orb06	10	10	0.09	9.2	0.22	8.6	0.00	8.5	–	–	0.00	0.9
orb07	10	10	0.00	0.3	0.00	0.1	0.00	0.5	–	–	0.00	0.1
orb08	10	10	0.00	6.0	0.08	5.8	0.39	7.2	–	–	0.00	0.7
orb09	10	10	0.00	0.5	0.29	2.6	0.00	0.4	–	–	0.00	0.2
orb10	10	10	0.00	0.2	0.00	0.1	0.00	0.3	–	–	0.00	0.1
yn1	20	20	1.55	40.0	1.12	27.6	0.71	106.3	0.02	190	0.44	55.6
yn2	20	20	1.04	62.3	1.63	27.7	0.24	110.4	0.30	197	0.49	39.1
yn3	20	20	1.24	42.2	1.21	43.5	0.39	110.8	0.15	212	0.49	39.0
yn4	20	20	1.94	56.0	1.95	45.1	0.48	108.7	0.11	216	0.79	41.5
swv01	20	10	3.66	60.6	4.01	64.6	1.19	142.1	0.32	299	1.56	114.0
swv02	20	10	3.05	64.4	2.05	58.6	0.36	119.7	0.00	18	1.09	68.3
swv03	20	10	2.58	48.7	2.80	57.9	1.39	139.1	0.55	312	2.05	76.8
swv04	20	10	3.25	57.7	4.99	98.6	0.66	143.9	0.23	339	2.44	86.1
swv05	20	10	4.80	58.1	4.35	63.7	1.39	146.7	0.26	338	1.53	94.6
swv06	20	15	4.02	81.9	4.95	62.1	1.32	192.5	0.38	402	1.28	108.2
swv07	20	15	3.39	68.5	4.67	53.2	1.96	190.2	0.44	392	2.58	68.0
swv08	20	15	3.54	71.8	4.39	66.7	1.36	190.0	0.67	394	2.23	67.9
swv09	20	15	3.17	68.7	4.07	58.8	1.70	193.8	0.33	393	2.58	40.1
swv10	20	15	4.45	85.7	4.32	32.0	0.95	184.6	0.89	403	2.58	60.1
Average			1.11	29.83	1.28	28.69	–	–	–	–	0.56	28.62

Table 15 Comparing the performance of the P3H$_4$ with that of other metaheuristics with respect to %DEV$_{avg}$ and running time for the QAP

Instance	PASS		SAFI		NDS		ACO-GA/LS		DivTS		CPTS		P3H$_4$	
	%DEV$_{avg}$	T$_{avg}$	%DEV$_{avg}$	T$_{avg}$	%DEV$_{avg}$	T$_{avg}$	%DEV$_{avg}$	T$_{avg}$	%DEV$_{avg}$	T$_{avg}$	%DEV$_{avg}$	T$_{avg}$	%DEV$_{avg}$	T$_{max}$
sko42	0.000	1.8	0.000	0.6	0.000	0.3	0.000	0.7	0.000	4.0	0.000	5.3	0.000	1.8
sko49	0.007	5.7	0.005	6.9	0.000	3.0	0.056	7.6	0.008	9.6	0.000	11.4	0.017	3.0
sko56	0.033	9.6	0.026	6.7	0.021	5.1	0.012	9.1	0.002	13.2	0.000	21.0	0.015	4.9
sko64	0.051	15.1	0.039	10.8	0.008	8.0	0.004	17.4	0.000	22.0	0.000	42.9	0.005	8.4
sko72	0.111	15.7	0.089	13.4	0.067	9.9	0.018	70.8	0.006	38.0	0.000	69.6	0.027	14.7
sko81	0.156	11.2	0.096	12.3	0.086	17.6	0.025	112.3	0.016	56.4	0.000	121.4	0.044	27.4
sko90	0.146	12.9	0.155	18.4	0.080	30.4	0.042	92.1	0.026	89.6	0.000	193.7	0.022	51.2
els19	0.000	0.036	0.000	0.055	0.000	0.00	–	–	0.000	0.2	0.000	0.1	0.000	0.4
bur26d	0.000	0.002	0.000	0.018	0.000	0.01	–	–	0.000	0.5	0.000	0.4	0.000	0.6
nug30	–	–	–	–	–	–	0.000	0.3	0.000	1.2	0.000	1.7	0.000	0.8
ste36c	0.000	0.887	0.000	0.880	0.000	0.41	0.000	0.5	0.000	2.3	0.000	2.5	0.000	1.2
lipa50a	–	–	–	–	–	–	0.000	5.2	0.000	6.6	0.000	11.2	0.000	3.2
tai64c	–	–	–	–	–	–	0.000	0.0	–	–	0.000	20.6	0.000	8.4
tai20b	–	–	–	–	0.000	0.0	–	–	0.000	0.2	0.000	0.1	0.000	0.4
tai25b	–	–	–	–	0.000	0.0	–	–	0.000	0.5	0.000	0.4	0.000	0.6
tai30b	–	–	–	–	0.000	0.4	0.000	0.3	0.000	1.3	0.000	1.2	0.000	0.8
tai35b	–	–	–	–	0.000	0.6	0.000	0.3	0.000	2.4	0.000	2.4	0.000	1.1
tai40b	–	–	–	–	0.000	0.3	0.000	0.6	0.000	3.2	0.000	4.5	0.000	1.6
tai50b	–	–	–	–	0.040	4.7	0.000	2.9	0.000	8.8	0.000	13.8	0.000	3.2
tai60b	–	–	–	–	0.034	4.8	0.000	2.8	0.000	17.1	0.000	30.4	0.003	6.4
tai80b	–	–	–	–	0.174	14.7	0.000	60.3	0.006	58.2	0.000	110.9	0.011	25.6
tai20a	0.000	0.2	0.000	0.053	0.000	0.0	–	–	0.000	0.2	0.000	0.1	0.000	0.4
tai25a	0.041	0.7	0.000	0.196	0.000	0.2	–	–	0.000	0.6	0.000	0.3	0.000	0.6
tai30a	0.181	1.6	0.000	0.419	0.007	1.8	0.341	1.4	0.000	1.3	0.000	1.6	0.000	0.8
tai40a	0.851	8.4	0.303	6.345	0.713	2.2	0.593	13.1	0.222	5.2	0.148	3.5	0.373	1.6
tai50a	1.324	11.8	0.715	9.546	1.146	3.9	0.901	29.7	0.725	10.2	0.440	10.3	0.614	3.2
tai60a	1.458	9.8	0.685	7.302	1.287	6.7	1.068	58.5	0.718	25.7	0.476	26.4	0.853	6.4
tai80a	1.524	17.9	0.987	14.601	1.459	15.7	1.178	152.2	0.753	52.7	0.570	94.8	0.971	25.6
Average									–	–	0.058	28.7	0.106	7.3

References

1. Adams J, Balas E, Zawack D (1988) The shifting bottleneck procedure for job shop scheduling. Manag Sci 391–401
2. Ahuja R, Orlin J, Tiwari A (2000) A greedy genetic algorithm for the quadratic assignment problem. Comput Oper Res 27(10):917–934
3. Aiex RM, Binato S, Resende MGC (2003) Parallel grasp with path-relinking for job shop scheduling. Parallel Comput 29(4):393–430. https://doi.org/10.1016/S0167-8191(03)00014-0. http://www.sciencedirect.com/science/article/pii/S0167819103000140
4. Alba E (2005) Parallel metaheuristics: a new class of algorithms, vol 47. Wiley, Hoboken
5. Aldous D, Vazirani U (1994) "Go with the winners" algorithms. In: 35th Annual symposium on foundations of computer science, 1994 proceedings, pp 492–501. https://doi.org/10.1109/SFCS.1994.365742
6. Amirghasemi M (2021) An effective decomposition-based stochastic algorithm for solving the permutation flow-shop scheduling problem. Algorithms 14(4). https://doi.org/10.3390/a14040112. https://www.mdpi.com/1999-4893/14/4/112
7. Amirghasemi M, Zamani R (2014) A synergetic combination of small and large neighborhood schemes in developing an effective procedure for solving the job shop scheduling problem. SpringerPlus 3(1):1–15. https://doi.org/10.1186/2193-1801-3-193
8. Amirghasemi M, Zamani R (2015) An effective asexual genetic algorithm for solving the job shop scheduling problem. Comput Ind Eng 83:123–138. https://doi.org/10.1016/j.cie.2015.02.011. http://www.sciencedirect.com/science/article/pii/S0360835215000686
9. Amirghasemi M, Zamani R (2017) An effective evolutionary hybrid for solving the permutation flowshop scheduling problem. Evol Comput 25(1):87–111. https://doi.org/10.1162/EVCO_a_00162
10. Amirghasemi M, Zamani R, Voß S (2018) An effective structural iterative refinement technique for solving the quadratic assignment problem. In: Cerulli R, Raiconi A (eds) Computational logistics. Springer International Publishing, Cham, pp 446–460

11. Amirghasemi M, Zamani R, Voß S (2019) Developing an effective decomposition-based procedure for solving the quadratic assignment problem. In: Paternina-Arboleda C (ed) Computational logistics. Springer International Publishing, Cham, pp 297–316
12. Applegate D, Cook W (1991) A computational study of the job-shop scheduling problem. ORSA J Comput 3(2):149–156
13. Babaee Tirkolaee E, Goli A, Weber GW (2020) Fuzzy mathematical programming and self-adaptive artificial fish swarm algorithm for just-in-time energy-aware flow shop scheduling problem with outsourcing option. IEEE Trans Fuzzy Syst 28(11):2772–2783. https://doi.org/10.1109/TFUZZ.2020.2998174
14. Baluja S, Caruana R (1995) Removing the genetics from the standard genetic algorithm. In: Prieditis A, Russel S (eds) Twelfth international conference on machine learning. San Francisco, Morgan Kaufmann Publishers, pp 38–46
15. Bertsekas D, Tsitsiklis J, Wu C (1997) Rollout algorithms for combinatorial optimization. J Heuristics 3(3):245–262
16. Bozejko W, Wodecki M (2002) Solving the flow shop problem by parallel tabu search. In: Proceedings. International conference on parallel computing in electrical engineering, pp 189–194. https://doi.org/10.1109/PCEE.2002.1115237
17. Bozejko W, Wodecki M (2004) Parallel genetic algorithm for the flow shop scheduling problem. In: Wyrzykowski R, Dongarra J, Paprzycki M, Waśniewski J (eds) Parallel processing and applied mathematics. Springer, Berlin, pp 566–571
18. Burkard RE, Karisch SE, Rendl F (1997) Qaplib–a quadratic assignment problem library. J Glob Optim 10(4):391–403
19. Carlier J (1982) The one-machine sequencing problem. Eur J Oper Res 11(1):42–47
20. Cecilia JM, Garcia JM, Ujaldon M, Nisbet A, Amos M (2011) Parallelization strategies for ant colony optimisation on GPUs. In: 2011 IEEE International symposium on parallel and distributed processing workshops and phd forum, pp 339–346. https://doi.org/10.1109/IPDPS.2011.170
21. Chapman B, Jost G, Van Der Pas R (2008) Using OpenMP: portable shared memory parallel programming, vol 10. MIT Press
22. Clerc M, Kennedy J (2002) The particle swarm-explosion, stability, and convergence in a multidimensional complex space. IEEE Trans Evol Comput 6(1):58–73
23. Crainic T (2019) Parallel metaheuristics and cooperative search. Springer International Publishing, Cham, pp 419–451. https://doi.org/10.1007/978-3-319-91086-4_13
24. Crainic TG, Toulouse M (2010) Parallel meta-heuristics. Springer, pp 497–541
25. De Jong KA (2006) Evolutionary computation: a unified approach. MIT Press. http://books.google.com.au/books?id=OIRQAAAAMAAJ
26. Derrac J, García S, Molina D, Herrera F (2011) A practical tutorial on the use of nonparametric statistical tests as a methodology for comparing evolutionary and swarm intelligence algorithms. Swarm Evol Comput 1(1):3–18. https://doi.org/10.1016/j.swevo.2011.02.002. https://www.sciencedirect.com/science/article/pii/S2210650211000034
27. Dorigo M, Gambardella LM (1997) Ant colony system: a cooperative learning approach to the traveling salesman problem. IEEE Trans Evol Comput 1(1):53–66
28. Drezner Z (2003) A new genetic algorithm for the quadratic assignment problem. Inf J Comput 15(3):320–330
29. Drezner Z, Hahn PM, Taillard É D (2005) Recent advances for the quadratic assignment problem with special emphasis on instances that are difficult for meta-heuristic methods. Ann Oper Res 139(1):65–94. https://doi.org/10.1007/s10479-005-3444-z
30. Falkenauer E, Bouffouix S (1991) A genetic algorithm for job shop. In: IEEE International conference on robotics and automation, 1991, proceedings, vol 1, pp 824–829. https://doi.org/10.1109/ROBOT.1991.131689
31. Fisher H, Thompson GL (1963) Probabilistic learning combinations of local job-shop scheduling rules. Industrial scheduling, pp 225–251
32. Garey M, Johnson D, Sethi R (1976) The complexity of flowshop and jobshop scheduling. Math Oper Res 117–129
33. Glover F (1989) Tabu search–part I. ORSA J Comput 1(3):190–206
34. Glover F (1998) A template for scatter search and path relinking. Lect Notes Comput Sci 1363:13–54
35. Hansen P, Mladenović N (2003) Variable neighborhood search. In: Glover F, Kochenberger GA (eds) Handbook of metaheuristics. Springer US, Boston, pp 145–184. https://doi.org/10.1007/0-306-48056-5_6
36. Holland JH (1975) Adaptation in natural and artificial systems. University of Michigan Press, Michigan
37. Iyer S, Saxena B (2004) Improved genetic algorithm for the permutation flowshop scheduling problem. Comput Oper Res 31(4):593–606
38. James T, Rego C, Glover F (2009a) A cooperative parallel tabu search algorithm for the quadratic assignment problem. Eur J Oper Res 195(3):810–826. https://doi.org/10.1016/j.ejor.2007.06.061. http://www.sciencedirect.com/science/article/pii/S0377221707011058
39. James T, Rego C, Glover F (2009b) Multistart tabu search and diversification strategies for the quadratic assignment problem. IEEE Trans Syst Man Cybern Part A: Syst Hum 39(3):579–596
40. Joshi SK, Bansal JC (2020) Parameter tuning for meta-heuristics. Knowl-Based Syst 189:105094. https://doi.org/10.1016/j.knosys.2019.105094. https://www.sciencedirect.com/science/article/pii/S0950705119304708
41. Lawrence S (1984) Resource constrained project scheduling: an experimental investigation of heuristic scheduling techniques (supplement). Report Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh
42. Lee Y, Orlin J (1993) Quickmatch: a very fast algorithm for the assignment problem. Report, Massachusetts Institute of Technology, Sloan School of Management (Report number: WP#3547-93)
43. Li Y, Pardalos P, Resende M (1994) A greedy randomized adaptive search procedure for the quadratic assignment problem. Quadratic Assignment and Related Problems 16:237–261
44. Lourenco H, Martin O, Stützle T (2003) Iterated local search. Handbook of metaheuristics, pp 320–353
45. Nagata Y, Ono I (2018) A guided local search with iterative ejections of bottleneck operations for the job shop scheduling problem. Comput Oper Res 90:60–71. https://doi.org/10.1016/j.cor.2017.09.017. https://www.sciencedirect.com/science/article/pii/S0305054817302460
46. Nawaz M, Enscore E, Ham I (1983) A heuristic algorithm for the m-machine n-job flow-shop sequencing problem. Omega 11(1):91–95
47. Nowicki E, Smutnicki C (1996) A fast tabu search algorithm for the permutation flow-shop problem. Eur J Oper Res 91(1):160–175
48. Nowicki E, Smutnicki C (2005) An advanced tabu search algorithm for the job shop problem. J Sched 8(2):145–159. https://doi.org/10.1007/s10951-005-6364-5
49. Osman IH, Potts CN (1989) Simulated annealing for permutation flow-shop scheduling. Omega 17(6):551–557. https://doi.org/10.1016/0305-0483(89)90059-5. http://www.sciencedirect.com/science/article/pii/0305048389900595
50. Pospichal P, Jaros J (2009) GPU-based acceleration of the genetic algorithm. GECCO competition

51. Rajendran C, Ziegler H (2004) Ant-colony algorithms for permutation flowshop scheduling to minimize makespan/total flowtime of jobs. Eur J Oper Res 155(2):426–438. https://doi.org/10.1016/S0377-2217(02)00908-6. http://www.sciencedirect.com/science/article/pii/S0377221702009086
52. Ravetti MG, Riveros C, Mendes A, Resende MGC, Pardalos PM (2012) Parallel hybrid heuristics for the permutation flow shop problem. Ann Oper Res 199(1):269–284. https://doi.org/10.1007/s10479-011-1056-3
53. Reeves C, Yamada T (1998) Genetic algorithms, path relinking, and the flowshop sequencing problem. Evol Comput 6(1):45–60
54. Rinnoy Kan A (1976) Machine sequencing problem: classification complexity and computation. The Hague, Martinus Nijhoff
55. Röck H (1984) The three-machine no-wait flow shop is np-complete. J ACM (JACM) 31(2):336–345
56. Ruiz R, Stützle T (2007) A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem. Eur J Oper Res 177(3):2033–2049. https://doi.org/10.1016/j.ejor.2005.12.009. http://www.sciencedirect.com/science/article/pii/S0377221705008507
57. Ruiz R, Maroto C, Alcaraz J (2006) Two new robust genetic algorithms for the flowshop scheduling problem. Omega 34(5):461–476. https://doi.org/10.1016/j.omega.2004.12.006. http://www.sciencedirect.com/science/article/pii/S0305048305000174
58. Sevkli M, Aydin ME (2007) Parallel variable neighbourhood search algorithms for job shop scheduling problems. IMA J Manag Math 18(2):117–133. https://doi.org/10.1093/imaman/dpm009. http://imaman.oxfordjournals.org/content/18/2/117.abstract
59. Stöppler S, Bierwirth C (1992) The application of a parallel genetic algorithm to the n/m/p/c max flowshop problem. In: Fandel G, Gulledge T, Jones A (eds) New directions for operations research in manufacturing. Springer, Berlin, pp 161–175. Book section 10. https://doi.org/10.1007/978-3-642-77537-6_10
60. Storer RH, Wu SD, Vaccari R (1992) New search spaces for sequencing problems with application to job shop scheduling. Manag Sci 38(10):1495–1509
61. Stützle T (1998) Applying iterated local search to the permutation flow shop problem. Technical Report AIDA-98-04. TU Darmstadt, FG Intellektik
62. Taillard E (1991) Robust taboo search for the quadratic assignment problem. Parallel Comput 17(4–5):443–455. https://doi.org/10.1016/S0167-8191(05)80147-4. http://www.sciencedirect.com/science/article/pii/S0167819105801474
63. Taillard E (1993) Benchmarks for basic scheduling problems. Eur J Oper Res 64(2):278–285
64. Taillard ED (1994) Parallel taboo search techniques for the job shop scheduling problem. ORSA J Comput 6(2):108–117
65. Talbi EG (2006) Parallel combinatorial optimization, vol 58. Wiley, Hoboken
66. Talbi EG, Roux O, Fonlupt C, Robillard D (2001) Parallel ant colonies for the quadratic assignment problem. Futur Gener Comput Syst 17(4):441–449. https://doi.org/10.1016/S0167-739X(99)00124-7. http://www.sciencedirect.com/science/article/pii/S0167739X99001247
67. Talukdar S, Murthy S, Akkiraju R (537) Asynchronous teams. Springer US, Boston. https://doi.org/10.1007/0-306-48056-5_19
68. Tasgetiren MF, Liang YC, Sevkli M, Gencyilmaz G (2007) A particle swarm optimization algorithm for makespan and total flowtime minimization in the permutation flowshop sequencing problem. Eur J Oper Res 177(3):1930–1947. https://doi.org/10.1016/j.ejor.2005.12.024. http://www.sciencedirect.com/science/article/pii/S0377221705008453
69. Tosun U, Dokeroglu T, Cosar A (2013) A robust island parallel genetic algorithm for the quadratic assignment problem. Int J Prod Res 51(14):4117–4133. https://doi.org/10.1080/00207543.2012.746798
70. Vallada E, Ruiz R (2009) Cooperative metaheuristics for the permutation flowshop scheduling problem. Eur J Oper Res 193(2):365–376. https://doi.org/10.1016/j.ejor.2007.11.049. http://www.sciencedirect.com/science/article/pii/S0377221707011253
71. Van Hentenryck P, Michel L (2009) Constraint-based local search. The MIT Press
72. Van Laarhoven P, Aarts E, Lenstra J (1992) Job shop scheduling by simulated annealing. Oper Res 40(1):113–125
73. Whitley D (1994) A genetic algorithm tutorial. Stat Comput 4(2):65–85
74. Whitley D, Rana S, Heckendorn RB (1999) The island model genetic algorithm: on separability, population size and convergence. J Comput Inf Technol 7:33–48
75. Wodecki M, Bozejko W (2002) Solving the flow shop problem by parallel simulated annealing. In: Wyrzykowski R, Dongarra J, Paprzycki M, Waśniewski J (eds) Parallel processing and applied mathematics, Lecture Notes in Computer Science, vol 2328. Springer, Berlin, pp 236–244. Book section 26. https://doi.org/10.1007/3-540-48086-2_26
76. Yamada T, Nakano R (1992) A genetic algorithm applicable to large-scale job-shop problems. Parallel Problem Solving from Nature 2:281–290
77. Zamani R, Amirghasemi M (2020) A self-adaptive nature-inspired procedure for solving the quadratic assignment problem. In: Khosravy M, Gupta N, Patel N, Senjyu T (eds) Frontier applications of nature inspired computation. Springer, Singapore, pp 119–147. https://doi.org/10.1007/978-981-15-2133-1_6
78. Zhang CY, Li P, Rao Y, Guan Z (2008) A very fast TS/SA algorithm for the job shop scheduling problem. Comput Oper Res 35(1):282–294. https://doi.org/10.1016/j.cor.2006.02.024. http://www.sciencedirect.com/science/article/pii/S0305054806000670, part Special Issue: Applications of {OR} in Finance
79. Zhu W, Curry J, Marquez A (2009) SIMD Tabu search for the quadratic assignment problem with graphics hardware acceleration. Int J Prod Res 48(4):1035–1047. https://doi.org/10.1080/00207540802555744
80. Zobolas G, Tarantilis CD, Ioannou G (2009) Minimizing makespan in permutation flow shop scheduling problems using a hybrid metaheuristic algorithm. Comput Oper Res 36(4):1249–1267

Mehrdad Amirghasemi received his M.Sc. in Intelligent Systems Design from Chalmers University of Technology, Gothenburg, Sweden and his Ph.D. in Computing and Information Technology (operations research) from the University of Wollongong, New South Wales, Australia. He is currently a Research Fellow at the SMART Infrastructure Facility, University of Wollongong, with his current research interest being the design and analysis of meta-heuristic algorithms for multiple problems in operations research, supply chain and logistics, IoT and cloud computing. He has published in multiple highly ranked journals such as Evolutionary Computation, Computers and Industrial Engineering, and Annals of Operations Research.