Heuristic Optimization Platform For Meta And Hyper-Heuristic Solving Of Combinatorial And Continuous Problems

Jon-Paul Boyd
School of Computer Science and Informatics
De Montfort University
United Kingdom

Abstract— This report presents a Python-based Heuristic Optimization Platform (HOP) to generalize the optimization of both combinatorial and continuous search problems using a variety of well-known single and multiple particle metaheuristic methods. A hyper-heuristic feature dynamically switches between low-level metaheuristics to deterministically use the most appropriate according to current problem state, or select one stochastically. Taillard's published FSSP problem instances were used to benchmark combinatorial optimisation, while the continuous Rastrigin problem was added to evaluate continuous optimization. Numerical testing results show several optimizer configurations are able to generalise across different problem types and still achieve satisfactory results, especially the hyper-heuristic combining Differential Evolution and Evolution Strategy methods. Recommendations include resolving the "*flyaway*" velocity issue when optimising combinatorial with Particle Swarm, further fine tuning of the hyper-heuristic choice function, and deployment to Cloud for scaling computing resources on demand to allow optimisation of larger problem instances.

Index – Optimization, Metaheuristics, Hyper-Heuristics

I. INTRODUCTION

All companies involved in manufacturing processes are concerned with the efficiency of limited production resources to maximise profit, meet throughput volume and timeline objectives, and support management decision making. However, as the number of jobs and machines involved in these processes increases, the complexity of optimal production scheduling becomes such that it cannot be solved manually. Within the domain of Operations Research (OR), which concerns itself with the application of mathematical models and analytical methods to support organizations address business operational challenges, the discipline of optimization continues to expand a family of algorithms that propose scheduling solutions.

The Flow Shop Scheduling Problem (FSSP) is a well-known combinatorial optimization problem where job operations must be processed on a set of machines with the objective of finding the permutation of jobs that minimises the complete processing time of the complete manufacturing operation, known as the makespan. Solutions look to optimize the production process and therefore the supply chain as a whole. Any solution proposal must adhere to specific constraints, namely that each job operation must be processed in order and a machine can only process one job operation at a time.

The FSSP problem is $n!$ Given this factorial nature, it becomes prohibitive calculating the makespan for all possible permutations. Taking a 20 job by 10 machine ($20j$x$10m$) scenario as an example, there are 2,432,902,008,176,640,000 possible permutations of job sequence. Further illustrating the challenge, solving a small $10j$x$10m$ benchmark instance published by Fisher and Thompson in 1963 took 23 years to solve [1]. As the author of [2] states, "*To find a schedule that minimizes the makespan, or one that minimizes the sum of completion times, was proved to be strongly NP-hard in the 70's, even for severely restricted instances*". It should therefore be clear that with such a large search space, any exacting, deterministic algorithmic method such as branch and bound, which look to "*find exact solutions for a wide array of optimization problems*" [3] and "*uses a tree search strategy to implicitly enumerate all possible solutions to a given problem*" [3], will be extremely computationally expensive.

Metaheuristic algorithms can be applied to scheduling problems to propose solutions that may only be near-optimal, but considered "*good enough*", at reasonable computational cost. As Caraffini et al. elaborates, "*Over the past decades, computer scientists have designed a multitude of these types of algorithms for addressing real-world problems where an exact approach is almost never applicable. These methods, known as metaheuristics, do not offer guarantees regarding the convergence, but still are capable to detect high quality solutions that can be of great interest for engineers and practitioners*" [4]. Inspired by Caraffini's SOS (Stochastic Optimisation Software) solution [5], written in Java for development and benchmarking of optimisation algorithms, this study presents HOP (Heuristic Optimisation Platform), a Python-based framework to address the real-world problem of scheduling optimisation. 5 nature-inspired metaheuristic algorithms including Simulated Annealing (SA), Genetic Algorithm (GA), Particle Swarm Optimisation (PSO), Differential Evolution Algorithm (DEA) and Evolution Strategy (ES) are implemented and evaluated using 3 benchmark instances from Taillard's FSSP problem suite [6].

A guiding principle in the design of HOP is the concept of a domain barrier which separates problems from heuristics such that none of the implemented algorithms are specifically coded for scheduling optimisation and therefore can be considered general purpose. As the No Free Lunch Theorems (NFLT) presented by Wolpert and Macready [7] suggest a program-tailored metaheuristic will show higher overall Avg performance, HOP additionally provides a novel hyper-heuristic feature which places the developed metaheuristic algorithms into a low-level heuristic (LLH) pool available for on-line consumption by the higher level heuristic during problem solving. This gives the framework a hybrid heuristic capability where exploitive local search SA can be combined with a more explorative, population-based method like PSO, extending the generality in which the framework can operate. Further supporting the guiding principle of problem and heuristic separation, HOP can also solve continuous problems with the same general metaheuristic pool without code change, tested using Rastrigin's function [8].

This paper is organized as follows. Section II provides further detail on the FSSP combinatorial and Rastrigin continuous problems used to benchmark HOP, while Section III presents the framework's architectural design and implementation. Section IV details the methodology behind the experimental design. Section V presents and discusses testing results, followed by Section VI which concludes with final thoughts.

II. The Problems

A. Formulation of the Flow Shop Scheduling Problem

The problem can be defined as follows. There are $M_{1,\dots,}M_m$ machines and $J_{1,\dots}J_n$ jobs. Each job J_j is a set of m machine processing time operations O_{jm}. The precedence constraint $O_{jm} \rightarrow O_{jm+1}$dictates that for a given job, operation 1 must be processed on machine 1 before proceeding to process operation 2 on machine 2. A feasible solution is one with a permutation of non-negative integer job numbers.

The objective is ordering the jobs in a sequence that minimises the makespan. An example solution optimised and visualisation by HOP for a 20*j*x5*m* problem is shown in Fig. 1. The visualisation is from a machine perspective, with the Gantt chart legend indicating the job sequence. Only when the first operation for job 14 completes on M_0 can both the second operation commence on $M_1$and the first operation for job 15 on M_0. As the second operation for job 15 needs to wait until the first operation completes, we can already see M_1 idle time where the machine is doing nothing.

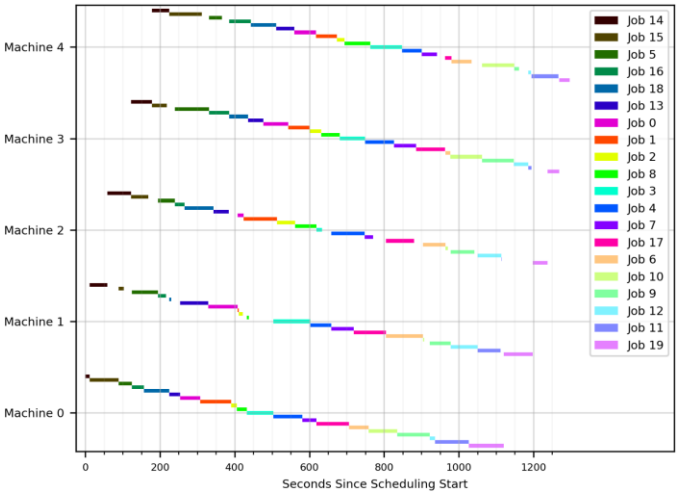

Fig. 1 FSSP 20*j*x5*m* problem optimised and visualised by HOP

B. Formulation of the Rastrigin Function

In contrast to the combinatorial FSSP, Rastrigin's function is continuous, and highly multi-modal with many local minima. HOP implements this testbed function in 2 dimensions as per the formula illustrated in Fig. 2.

$$f(x) = 10n + \sum_{i=1}^{n} \left[x_i^2 - 10\cos(2\pi x_i) \right], \; -5.12 \leq x_i \leq 5.12$$

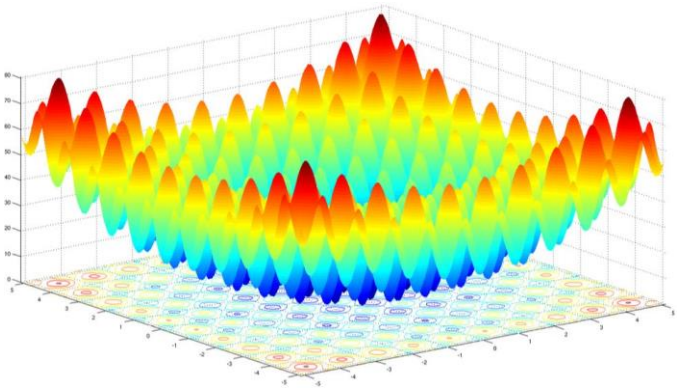

Fig. 2 Rastrigin's Function Formula & Plot

III. Architecture Design & Implementation

The general architecture of HOP is presented in Fig. 3. It is composed of 3 sub-architectural elements: heuristics, the domain barrier, and problems. "*Heuristics*" encapsulates the problem-independent implementation of the individual, nature-inspired algorithms and hyper-heuristic solution. The coded representation of FSSP and Rastrigin, with associated benchmark instances, are grouped under "*Problems*".

Separating "*Heuristics*" and "*Problems*" is the domain barrier and home of the heuristics manager (HM) which orchestrates the optimisation process. The platform incorporates a number of explicit software design choices. These including using an object-orientated programming paradigm to abstract FSSP and Rastrigin as instances of the problem class, and the implemented heuristics as instances of the optimizer class. This facilitates sharing of generic, "*superclass*" code, minimising maintenance and speeding up any future implementation of new problems and optimizers. Further, data transfer across the barrier occurs only via a "job *specification*" object. YAML files support code-free change by maintaining general runtime parameters in addition to specific configurations for optimizers and problems.

The processing flow of HOP is as follows. The platform is launched by running *main.py* (1), which first creates a timestamped log file and results folder (2). Control is handed over to the HM (*heuristics_manager.py*) which coordinates all optimization processes (3). The first responsibility of the HM is to upload the 3 configuration files (4) from which it determines the implemented problems (pid), optimisers (oid) and benchmarks (bid) (5). A job specification class object defined in *hopjob.py* is then created for each combination of pid/oid/bid (6). This persists runtime information including computational budget allowance, henceforth referred to as budget, and tracks global and individual run best candidate solution and associated fitness value. Instances of configured classes and methods objects for pid, oid, candidate generator, variator and crossover are determined and created at this point, allowing lower level metaheuristics to focus only on optimization, consuming the generator or crossover method defined in the available job spec. This approach further enforces domain barrier separation of concerns by placing the responsibility of deciding which optimization support methods to call with the HM. An added, important benefit is reduced complexity in each metaheuristic implementation.

Each job spec is added to a job list (7). Note this list can include jobs where the optimizer is configured as disabled but consumed within a low-level heuristic (LLH) pool by a hyper-heuristic configuration. During the main loop, tracking attributes like fitness trend, run best and population are reset in a pre-processing step (8) before each active pid/oid/bid specification is executed (9) a number of times for trusted statistical analysis. Upon completion of the optimizer, the run best candidate fitness is compared with global best (10) before further optimizer run post-processing generates a line plot and *.csv* file for the given run fitness trend (11).

When all execution runs for a specification are completed, the HM executes the current problem class *post_processing* method for any specific requirements, in the case of FSSP generating the global best solution Gantt chart. When all job specifications for each optimized problem are complete, a statistical summary report and global best fitness trend plot is generated (12).

What now follows is a closer look at a selected set of HOP components that warrant additional detail.

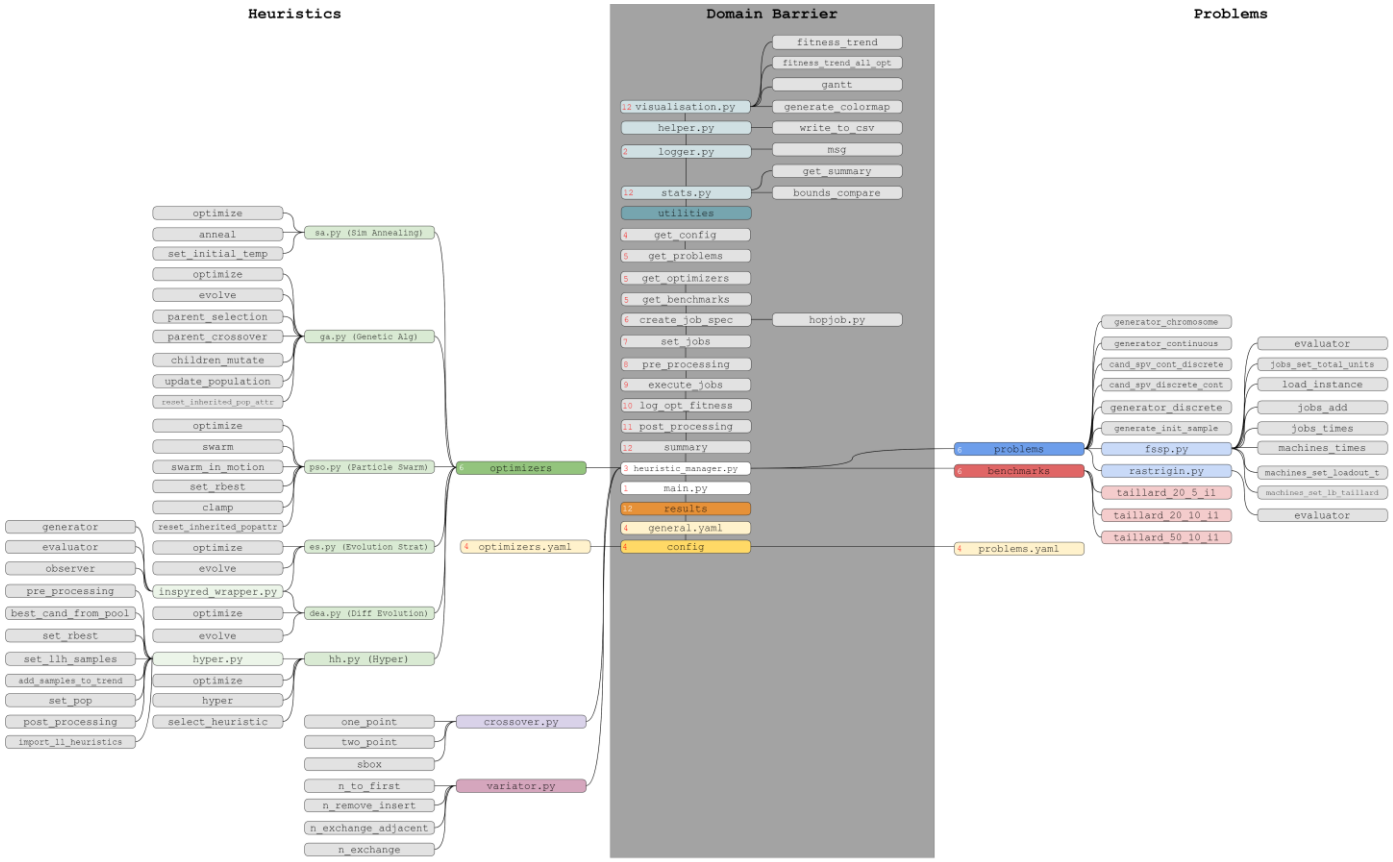

Fig. 3 HOP Architecture

A. *Optimization Experiment Configuration*

The platform comprises 3 configuration files. The first, *general.yaml*, defines the number of runs per optimizer, budget base, and bit computing size which determines the number of genes when generating chromosome-based candidate solutions.

Configuration for the FSSP and Rastrigin problems are defined in *problems.yaml*. An enabled flag allows problems to be toggled on/off. The problem type is specified; combinatorial for FSSP and continuous for Rastrigin, which helps support determination of correct candidate generator method. Lower and upper bounds for each problem are specified; [0, nmax] for FSSP, and [-5.12, 5.12] for Rastrigin. These bounds are used when generating new candidate solutions and decoding binary representations, used in GA, back to real numbers.

Configuration for all available metaheuristic variants is maintained in *optimizers.yaml*. The metaheuristic with id SA-N-E in the example below configures a low-level SA optimizer, specifying the use of a *discrete* generator when handling combinatorial problems otherwise a *continuous* generator. An *n_exchange* variator will be used to perturb candidate solutions, while an initial candidate sample is required to determine starting temperature. The reheat feature, which restarts the annealing process by resetting starting temperature but continuing with run best candidate solution is disabled.

```
SA-N-E:
    enabled: True
    description: Simulated Annealing with exchange variator
    type: low
    optimizer: SA
    generator_comb: discrete
    generator_cont: continuous
    variator: n_exchange
    initial_sample: True
    lb: null
    ub: null
    reheat: False
```

For comparison, an example hyper-heuristic model configuration makes available the PSO, GA with SBOX crossover, and SA with remove-insert variator available in the LLH pool. Each LLH component is executed 3 times to provide the Q table tracking metaheuristic performance with sample candidates and their respective fitness values. A budget of 0.01% of the budget will be allocated to each component in the sampling run, while 0.05% of total is given to the metaheuristic component selected by the hyper-heuristic choice function. Decay parameters determine the probability of selecting a metaheuristic other than the best performing according to the Q table.

```
HH-GA-N-E-SBOX-PSO-SA-N-RI:
    enabled: True
    description: Hyper heuristics - GA & PSO & SA
    type: hyper
    optimizer: HH
    low_level_selection_pool:
        - PSO
        - GA-N-E-SBOX
        - SA-N-RI
    decay: 0.8
    decay_coeff: 0.99
    llh_sample_runs: 3
    llh_sample_budget_coeff:  0.01
    llh_budget_coeff: 0.05
```

B. *Heuristic Manager*

The HM implemented in *heuristic_manager.py* is primarily responsible for the heavy lifting of parsing experiment configuration into a set of job specifications with instantiated problem and optimizer objects, hooks to generator, variator and crossover methods, and tracking of budget and optimizer performance history. It executes each enabled LLH, and for hyper-heuristics, provides a runtime specification of each LLH in their configured selection pool. Assigning these tasks to a central, core component greatly simplifies the implementation and concerns of the optimizers themselves. Note HM explicitly seeds the standard Python and numpy package generators over multiple executions and environments.

C. *Problem Implementation*

The FSSP and Rastrigin problems are implemented in *fssp.py* and r*astrigin.py* respectively, inheriting from the *Problem* superclass defined in *problem.py*. This superclass defines methods for generating candidate solutions in discrete, continuous or chromosome form, as well as methods transforming discrete candidates to continuous and vice versa. The main method of the problem-specific class itself is the

evaluator method in which an objective function evaluates a candidate solution and returns a fitness value. For FSSP this is the makespan, or total manufacturing time of all job operations. Note the rather complex FSSP objective function was verified correct using published best-known permutations. In HOP, the Rastrigin objective function is two-dimensional, with the global minimum $f(x) = 0$ at $x(0, 0)$. As shown on Fig. 3, the FSSP class implements additional methods for domain-specific insight into proposed candidate solutions, such as machine idle and loadout times.

D. Low-Level Heuristic Implementation

A number of nature inspired metaheuristic algorithms, categorized as low-level to distinguish them from hyper-heuristic variants, have been implemented in HOP. In the case of the FSSP, the objective of the metaheuristic is to search for and propose candidate solutions that minimize makespan, and for Rastrigin, find the minimum of the function itself, ideally the one global minimum centrally located at $x(0, 0)$ amongst many equally distributed local minima. They all work by modifying or inheriting parts of an existing, complete solution to generate a new candidate solution, in the case of FSSP a new permutation of job order sequence. The new solution is evaluated for fitness with the problem-specific objective function. With the exception of SA, the new solution is only accepted and promoted to current run best if the fitness value is better (lower) than the previous best. These algorithms are now described in more detail.

1) Simulated Annealing

Fully hand-crafted in *sa.py*, SA [9] is a classic metaheuristic inspired by the natural phenomenon of heated metal or glass being cooled in a controlled fashion to reduce internal stress and eventually crystalize. Starting temperature takes the 60th percentile of an initial sample fitness. It performs local neighbourhood search, in the case of FSSP by perturbing job order according to configured variator type or generating a new 2-dimensional candidate solution for the continuous Rastrigin problem.

Within HOP, SA is a unique metaheuristic in 2 ways. Firstly, a stochastic mechanism allows it to be the only LLH to accept a solution worse than current best, helping it try to avoid being trapped in local minima, get "*over the hill*" and into a different area of the search space. During each iteration of the optimization process the starting temperature is cooled by a cooling rate co-efficient, with the probability of accepting a worse solution decreasing as the temperature drops. This stochastic mechanism calculates *loss* as current best fitness minus new candidate fitness, then calculates the probability of accepting a worse solution as the natural exponent of *loss* divided by current temperature. Testing showed it was necessary to cap loss at *0.3* to "*stabilize*" algorithm operation across both problem types.

The second unique aspect is early termination before allocated budget is exhausted, triggered by the temperature falling below a preconfigured threshold.

2) Genetic Algorithm

The GA, implemented in *ga.py*, is "*widely used as search techniques based on survival of fittest theory by Darwin*" [10], and is the first of two population-based algorithms fully custom developed in HOP that perform a global search with multiple candidates simultaneously. An initial population of random candidates forms a chromosome pool from which parents are selected to mate and form offspring. In this way the population evolves through a series of generations. Note that for the FSSP combinatorial problem, the genotype is represented by the job permutation, whereas in continuous problems the real values are binary encoded for manipulation.

With current configuration, 2 parents are selected using a method known as stochastic universal sampling (SUS) [11]. The intuition behind SUS is that the sum total fitness of the population forms a line in the range [0, 1]. The line is then divided into contiguous segments proportionate to the fitness value of each individual. As we are dealing with minimisation problems, the lower the fitness value the larger the segment. Pointers equal to the number of parents to be selected are equally spaced on the line, with point distance calculated as 1 divided by number of parents. Critically, the starting location of the first point on the line is a random number in the range [0, point distance]. If a segment has a pointer placed within it, the associated candidate is selected. This approach allows weaker individuals to breed and thereby ensure diversity in the population. A check on the sum total fitness equalling zero captures convergence, as is the case when optimizing the Rastrigin problem.

Crossover methods including one point, two point and sbox select genes from both parents to form new child individuals. Each child is then further mutated with one of several variator methods (again according to metaheuristic configuration) only if both parents are too similar. This similarity check compares genes from both parents, retaining order significance. If the percentage of matching genes exceeds a threshold, currently set at 80%, mutation takes place. This further ensures diversity in the population and prevents the optimizer from getting trapped in local minima. This situation occurs more frequently in smaller combinatorial problems, than say, 50 job Taillard FSSP instances and where continuous search transforms real numbers into 32-bit binary.

A new population is formed from both parents and their new-born children, with remaining individuals from the previous population discarded. As one of the parents is typically the fittest candidate from the previous population, this approach follows an elitist strategy. The evolutionary cycle continues until budget is exhausted or convergence terminates the process.

3) Particle Swarm Optimisation

The second population-based metaheuristic approach custom developed in HOP is PSO, first developed by Kennedy and Eberhart in 1995 [12]. As described by the authors of [13], "*PSO is based on the metaphor of social interaction and communication such as bird flocking and fish schooling. PSO is distinctly different from other evolutionary-type methods in a way that it does not use the filtering operation (such as crossover and/or mutation), and the members of the entire population are maintained through the search procedure so that information is socially shared among individuals to direct the search towards the best position in the search space*".

Implemented in *pso.py*, an initial random swarm "*curr*" is assembled. The swarm size is set in HM when compiling the job specification, with the problem dimensions *n* the basis for calculation. If the problem is combinatorial, swarm size is *n*2*, otherwise *n*3*. For an FSSP 20 job problem there would be 40 individuals, for the 2 dimensional Rastrigin problem 6 individuals.

PSO works in a continuous value space, so to address the challenge of evaluating candidates when optimising the discrete, combinatorial FSSP problem, a Smallest Position Value (SPV) heuristic [14] rebuilds a real value vector-based solution back to a discrete, feasible schedule permutation. It works by taking the index of the smallest position and assigns that as job 1, then the index of the next smallest value and assigns that as job 2, and so on.

Following candidate transformation, each individual is evaluated. Two snapshots of the initial swarm are taken. One labelled "*gbest*" is used to track the best location each candidate has visited with associated solution and fitness value. The second, labelled "*prev*", will persist a view of each swarm particle from the previous optimisation iteration, used to calculate the inertia towards an improved location. The fittest swarm candidate is set as "*rbest*" (run best).

The optimisation loop then begins. HOP implementation applies PSO with a global neighbourhood where each particle moves towards its previous best and that of the swarm leader, with this process now described.

A "*new*" swarm is created by perturbing all dimensions of each individual in the "*curr*" swarm with the combined sum of:

- **Inertia** where the individual is now plus a co-efficient of loss from "*prev*" location minus "*curr*" location
- **Local** loss from "*gbest*" location minus "*curr*" location multiplied by a local co-efficient and random [0, 1]
- **Global** loss from "*rbest*" leader minus "*curr*" location multiplied by a global co-efficient and random [0, 1]

The "*curr*" swarm is then stored in the "*prev*" swarm snapshot before the "*new*" swarm is copied to the "*curr*" swarm. A fitness evaluation of all candidates in "*curr*" takes place. If fitness improves for a candidate, it's particle is updated in "*gbest*". If any fitness improves upon "*rbest*" (optimizer run best), then "*rbest*" is updated with the fittest of all particles to become the new global swarm leader. This process continues until computational budget is exhausted.

4) Inspyred Python Package

The Inspyred package [15], developed in Python, makes available a large collection of bio-inspired algorithms that are readily consumable. The author of this paper selected the Differential Evolution Algorithm (DEA) and Evolution Strategy (ES) meta-heuristics from Inspyred to investigate the ease of integration of algorithms from other packages into HOP. Integration proved straight-forward. This capability allows the rapid extension of the LLH component pool both for single algorithm and hyper-heuristic search.

As per fully custom developed algorithms, class implementations in *dea.py* and *es.py* inherit from the same *Optimizer* superclass. This provides the integration point into the HM. An additional *inspyred_wrapper.py* component defines hooks that the Inspyred optimization engine executes when both generating and evaluating solutions, and observing optimization progress. It is here, in this wrapper, that HOP based, problem-specific evaluation calls to the FSSP and Rastrigin implementations are made, correct candidate solutions generated according to job specification, fitness trends updated, and budget consumed.

E. Generators

Generator functions provide metaheuristics with a single random candidate solution. In HOP 3 such generators are defined in *problem.py* (Fig. 3). Method *generator_discrete* creates a new discrete permutation with number of elements limited to problem dimension. Method *generator_continuous* generates *n* elements according to problem dimension, bound by lower and upper limits defined in configuration, for example Rastrigin [-5.12, 5.12]. Method *generator_chromosome* creates a random byte string with length defined by configuration parameter *bit_computing* currently set to 32 in *general.yaml*.

F. Variators

Variators support neighbourhood search in perturbing an existing candidate solution by one step. HOP provides 4 variator types, defined in *variator.py*, that are available when configuring optimizers:

- **n_exchange** swaps random positions *i* and *j*, for example swapping job 4 at the 2nd position with job 19 at the 7th position in an FSSP schedule permutation.
- **n_exchange_adjacent** swaps positions *i* with *i+1*
- **n_remove_insert** removes *i* and inserts at *j*
- **n_to_first** removes *i* and inserts at first position

All variators support perturbation of combinatorial candidates with a single permutation solution, and continuous problem solutions represented in *n* dimensions, for example in the 2-dimensional Rastrigin problem where there are 2 chromosome byte string elements to perturb.

G. Crossovers

Crossover functions support genetic evolution by inheriting features of parents when creating new child candidate solutions. 3 such functions have been developed for HOP, defined in *crossover.py* (Fig. 3). One point crossover is illustrated with a 10 job FSSP candidate solution in Fig. 4. Jobs before a random crossover point are taken from parent 1, with the remaining positions derived from parent 2, starting from the left. Only jobs not already present in the child are sourced from the second parent.

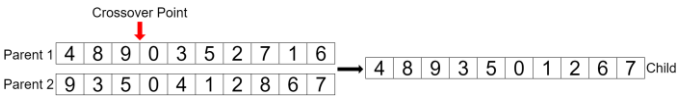

Fig. 4 One-point Crossover

Two-point crossover (Fig. 5) sets 2 random crossover points. Unique jobs to the left of position 1 and the right of position 2 are carried over from parent 1 to the child, with remaining positions to form a feasible solution sourced from parent 2.

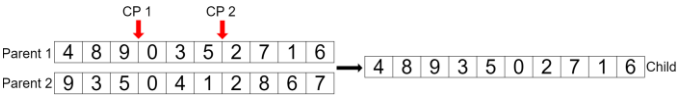

Fig. 5 Two-point Crossover

Hop also implements the SBOX crossover method, inspired by the authors of [16] and described as "*The operator SBOX (similar block order crossover) first copies blocks of common jobs contained in both parents into the offspring, where a block consists of two consecutive genes. Then a cut point is determined and jobs up to this point are taken from the first parent, while the missing jobs are copied into the empty genes in the relative order of the other parent.*". The operation is demonstrated by Fig. 6 with a binary chromosome example.

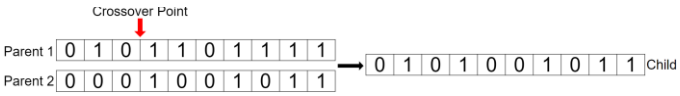

Fig. 6 SBOX crossover

H. Hyperheuristic Implementation

Hyper-heuristics have been described by Burke et al. as "*a set of methods that are broadly concerned with intelligently selecting or generating a suitable heuristic in a given situation. Hyper-heuristics can be considered as search methods that operate on lower-level heuristics or heuristic components*" [17]. HOP adopts this perspective with a hyper-heuristic framework implemented in *hh.py* (Fig. 3). Executed from the HM just like any other optimizer, it has a full runtime view of the LLH components available to it via job specifications. In essence, it is a simplified version of the HM and its *execute_jobs* method but adds a heuristic choice function to dynamically change the preference of LLH based on historic performance, with a stochastic twist.

The hyper-heuristic implements a Q (quality) table, which is initially populated with candidate solutions from all available LLH components. The purpose is initial exploration of the problem with a limited portion of the overall computational budget, set by configuration (tested with 0.01%), to sample the search space.

The optimisation loop then begins and continues until the full budget is exhausted. The design of the choice function heuristic could be described as SA for hyper-heuristics. It first selects the LLH responsible for the current fittest solution. However, this preference can be replaced if a decay factor, that decreases over time, is greater than random [0, 1]. Initially, sub-optimal heuristics have a good probability of being selected to explore the problem space, but as the decay reduces, the hyper-heuristic highly prefers the strongest LLH component for exploitation. The selected LLH is executed with a greater portion of the total budget to allow the metaheuristic to "*stretch it's search legs*" more than in the sampling step. It should be noted

that the HH "*warm starts*" the called LLH by either seeding its run best for single particle metaheuristics (SA), or pre-populating the initial population (PSA, GA, ES, DEA) with its fittest candidate in the Q table. On run completion, the Q table is updated with LLH run best. Runtime statistics are maintained for the number of times each component was selected, and how much they improved the run best fitness value by.

I. Logging

A log file is continually maintained through an experiment run, the detail it contains set by the logging level specified in main.py. It is an especially useful tool, invaluable during framework and heuristic development, testing and detailed analysis.

J. Summary Results

Upon completion of an experiment run, *stats.py* (Fig. 3) manages the collection of optimisation statistics including min, mean, max and stdev for fitness, and Wilcoxon ranksum pairwise comparison, writing out to both the log and summary *.csv* file. Plots showing the fitness trend for individual metaheuristics, and all for comparison purposes, is handled by *utilities.py* (Fig. 3). A post-processing method in each problem implementation allows problem-specific generation of analytics and visualisations, , in the case of FSSP the Gantt chart showing the proposed schedule for global best permutation (Fig. 1).

IV. Experimental Design

All development and experiments were carried out on an Intel i7-5820K clocked at 4.6Ghz, with 16GB RAM, running Windows 10 Professional (v18.3), Python 3.7.2 and PyCharm Professional 2018.3.7.

To ensure fair comparison between all heuristics, both meta and hyper, each were allocated the same budget base of 3000 (*general.yaml*) multiplied by the number of dimensions in the problem space. This implies a budget of 60000 for a 20 job FSSP problem and 6000 for Rastrigin. Each optimizer was executed 20 times, with the best fitness value for each run stored for later statistical analysis.

To test performance in solving the FSSP, 3 benchmark instances from the Taillard suite [6] were used – 20*j*x5*m*, 20*j*x10*m* and 50*j*x10*m*. The main evaluation metric for FSSP problems is makespan, the total time to complete all jobs in the schedule. The best minimum fitness, or makespan, generated by each optimizer is compared with Taillard's upper bound which is the best known makespan achieved for the given instance.

After initial testing of 23 optimizer configurations (*optimizers.yaml*), this list was pruned to the following 9:

SA-N-E	Simulated Annealing with exchange variator
DE	Differential Evolution (Inspyred)
ES	Evolution Strategy (Inspyred)
GA-N-E-1P	Genetic Algorithm with exchange adjacent variator and one-point crossover
GA-N-RI-SBOX	Genetic Algorithm with remove-insert variator and sbox crossover
PSO	Particle Swarm Optimisation
HH-SA-ALL	Hyper heuristic with pool of 4 Simulated Annealing Variants
HH-DE-ES	Hyper-heuristic with DE and ES (Inspyred)
HH-GA-N-E-SBOX-PSO-SA-N-RI	Hyper-heuristic with a Genetic Algorithm variant, SA variant and PSO

V. Results

Starting with the Taillard FSSP benchmark problems, metaheuristics performance is now analysed with the support of the numerical results presented in Tables I-III. As a generalisation, a lower stdev indicates the associated optimizers are better able to hone in on the fitter near-optimals. Regarding the smallest test instance (Table I), all optimizers are at maximum within 1.49% of Taillard's best known upper bound of 1278, with 5 of the 9 optimizers matching it, including 2 of the hyper-heuristic configurations and the integrated ES algorithm from Inspyred. PSO and SA-N-E are the most unstable, with highest variability in performance indicated by high stdev and fluctuating plot lines on Fig. 7. Regardless, PSO was also able to match best known upper bound, unlike SA-N-E. In fact PSO is responsible for the highest makespan of any optimizer, curiously attained from it's first run, after which it seems to stabilize (Fig. 7). This PSO first run poor performance is also present when tested in higher dimensional problem instances (Figs. 8, 9) and is to be investigated.

Typical SA oscillatory characteristics are present in the plot line for SA-N-E and suggested by the highest Avg improvement count. Given a budget of 60000 was allocated to it, SA-N-E last improved on avg iteration 3344, with 56656 iterations or 94.43% of budget without improving the makespan. This would indicate fine tuning of initial temperature, cooling rate and loss capping is required.

The hyper-heuristic configurations HH-DE-ES and HH-GA-N-E-SBOX-PSO-SA-N-RI were able to match Taillard best known upper bound despite recording the lowest improvement count of any optimizer. It is hypothesised the early sampling and choice function are effective in selecting a metaheuristic that fits the current problem state. This claim is further supported by their flat plotlines for the majority of the 20 run trend plot. Examining the logfile for HH-DE-ES, a typical run shows LLH component DE was executed 42 times with aggregated fitness improvement over initial sample seed of 18, while LLH ES was executed 52 times contributing an improvement of 42. Combined, the 2 components are able to support the hyper-heuristic in finding a permuation that evaluates to Taillards best known makespan and matches the best of other optimizers, whilst doing so at 0.05% of the standalone budget per metaheuristic execution. It's average compuational time in seconds (AvgCts) is 16s whilse ES standalone is 25s and DE 65s. Hyper-heuristic HH-GA-N-E-SBOX-PSO-SA-N-RI is at least 1s slower than any of its LLH components when tested standalone, however is more stable with smaller stdev.

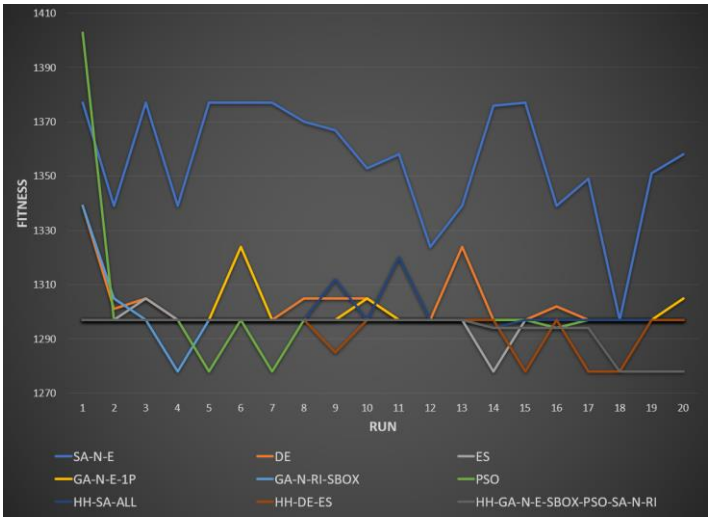

Fig. 7 Taillard FSSP 20 jobs 5 machines all optimizer fitness trend over 20 runs

Upon review of results from testing with larger problem instances, it is clear the expanded search space makes it both more difficult for the metaheuristics. Consistency degradades, with stdev generally increasing in line with instance size, and no match for Taillard's upper bound, with UB difference % > 0 in all cases. For the 20*j*x10*m* problem, GA-N-E-1P and GA-N-RI-SBOX attain best makespan (UB Diff % 0.25), albeit at a small cost to stability (stdev ~ ±15.5) compared with HH-DE-ES (UB Diff % 0.57, stdev ±9.655).

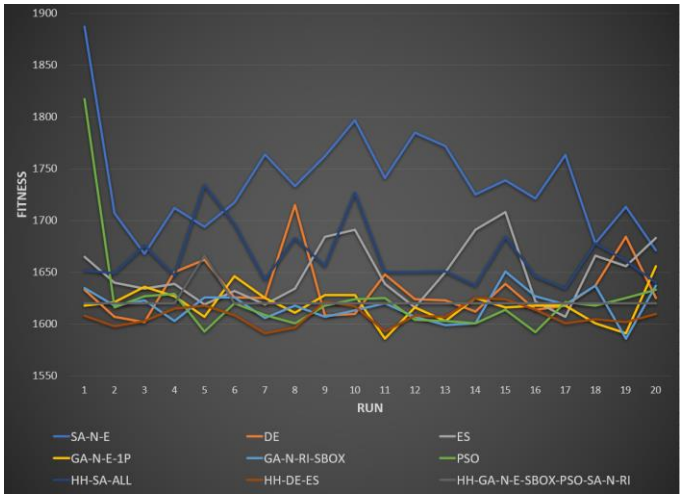

Fig. 8 Taillard FSSP 20 jobs 10 machines all optimizer fitness trend over 20 runs

PSO stability very clearly degrades as the dimensions increase, with StDev jumping to ±45.679 (Table II) then ±104.355 (Table III). The author theorises that in the specific scenario of combinatorial optimisation by PSO, where the current HOP implementation does not specify real-value lower and upper bounds, the swarm is out of control and flying outside of a reasonable search space. Further fine tuning of velocity co-efficients and clamping may mitigate this problem. Despite this issue it performs reasonably well as a standalone metaheuristic, achieving 3rd best makespsan for 20*j*x10*m* and 4th best for 50*j*x10*m*. As an LLH component in hyper-heuristic HH-GA-N-E-SBOX-PSO-SA-N-RI, the logs show PSO consistently contributing the most makepsan imporvement (e.g. for run 2, PSO executed 51 times imp. 294, GA-N-E-SBOX 26 times imp. 0, SA-N-RI 17 times imp. 113).

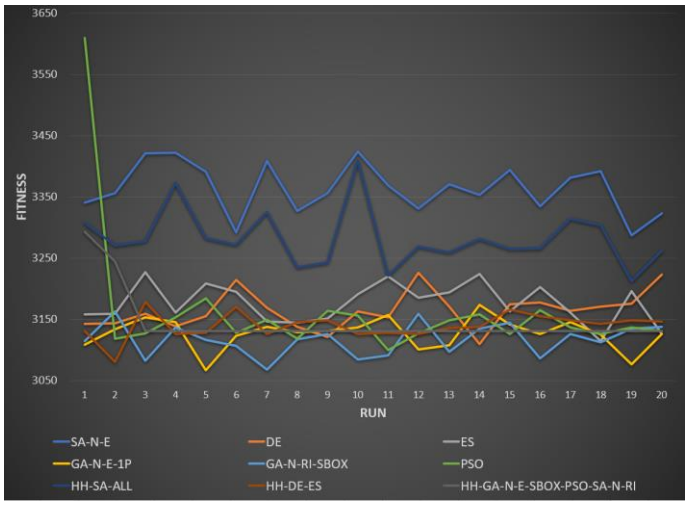

Fig. 9 Taillard FSSP 50 jobs 10 machines all optimizer fitness trend over 20 runs

Another trend is the stability of hyper-heuristic HH-DE-ES regardless of problem dimension. It is consistenly stable throughout the tested benchmarks (±7.032, ±9.655, ±20.544), where it's fitness trend exhibits small amplitude oscillations (Figs. 8, 9), and returns good makespan performance (UB Diff 0%, 0.57%, 1.82%). Indeed, for the 20*j*x10*m* and 50*j*x10*m* instances, hyper-heuristic HH-DE-ES performs better than either of its standlone components. A further consistent trend is DE being the slowest of all metaheuristics, taking minimum twice the time to complete an optimization run.

Moving to the Rastrigin benchmark problem, the results in Table IV show the standalone SA variant returns the poorest minimum makespan, and is therefore the worst performer in both problem domains. It is hypothesised that the single search mechanism stuggles to find the single global minimuma, either getting trapped in one of the many local minima or oscillating between them. Overall, 6 of the 9 optimizers successsfully minmised to global, with runtime performance for all sub-second. Once more, hyper-heuristic HH-DE-ES proves to be extremely stable, with stdev of ±0.0 and max fitness of 0.001. It is able to work well in both combinatorial and continuous problem domains across a variety of problem dimensions, albeit with slower runtime than some of the other metaheuristic variants.

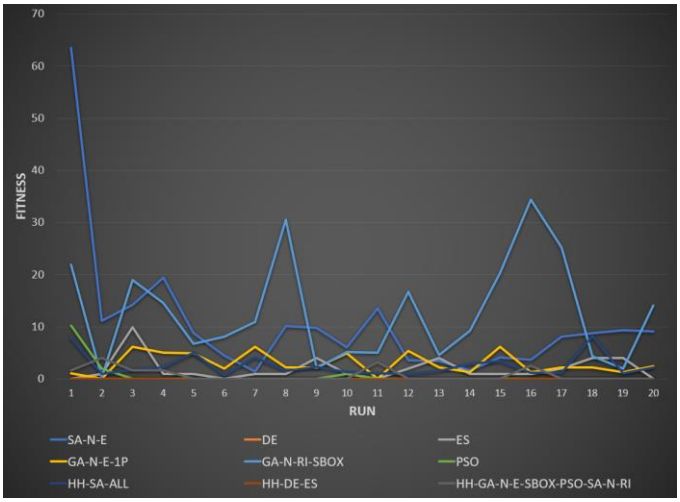

Fig. 10 Rastrigin 2-dimension problem all optimizer fitness trend over 20 runs

VI. CONCLUSION

This study designed and implemented HOP, a platform able to generalise the optimization of both combinatorial and continuous problems using a variety of well-known single and multiple particle metaheuristic approaches. A higher-level hyper-heuristic feature is able to consume available meta-heuristics, dynamically adapting to the current problem state and either selecting the most appropriate optimiser or further explore stochastically. In optimising FSSP combinatorial problems, performance was benchmarked against well-known, published makespan values. It achieved satisfactory results, especially at lower dimensions where Taillard's best known upper bound was matched. Unfortunately, no upper bounds were improved upon. In optimising the Rastrigin 2 dimensional problem, global minimum was found by several optimizers.

Further improvements might include the dynamic, real-time deterministic selection of variator and crossover methods according to performance history, or stochastically to introduce yet further diversity. There is always scope for improving the hyper-heuristic choice selector, perhaps initially by applying a short-term memory feature to "*age*" and disregard older fitness evaluations when selecting the metaheuristic component to execute. More experimentation with LLH budget allocation might further explain why some metaheuristics perform better as part of a hyper configuration.

It would be desirable to test with even larger FSSP problem instances, but unfortunately is beyond the capabilities of the current experiment setup, given the 50*j*x10*m* for 9 optimizers experiment took ~ 12 hours to complete. Hence, of great interest, would be to take HOP to a cloud-hosted environment, where the metaheuristics are deployed as FaaS (Functions as a Service) and required computing resources can scale with demand. All HOP code is available on GitHub [18].

TABLE I

TAILLARD FSSP 20 JOBS 5 MACHINES – OPTIMIZER RESULTS - BEST KNOWN MAKESPAN 1278

Optimizer	Makespan					Wilcoxon	Avg Iteration Last Imp	Budget No Imp %	Avg Imp Count	Avg Cts
	Min	Max	Avg	StDev	UB Diff %					
SA-N-E	1297	1377	1356	±21.348	1.49		3344	94.43	259	0.033
DE	1297	1339	1302	±10.452	1.49	-	4358	92.74	10	65.796
ES	**1278**	1305	1296	±4.577	0	-	7748	87.09	11	25.679
GA-N-E-1P	1297	1324	1299	±6.183	1.49	-	5156	91.41	9	4.902
GA-N-RI-SBOX	**1278**	1339	1298	±10.346	0	-	2937	95.1	10	5.199
PSO	**1278**	1403	1300	±24.244	0	-	7497	87.5	9	7.443
HH-SA-ALL	1294	1320	1298	±5.923	1.25	-	42597	29	6	3.461
HH-DE-ES	**1278**	1297	1293	±7.032	0	-	10440	82.6	2	16.496
HH-GA-N-E-SBOX-PSO-SA-N-RI	**1278**	1297	1293	±6.637	0	-	32837	45.27	2	8.204

TABLE II

TAILLARD FSSP 20 JOBS 10 MACHINES – OPTIMIZER RESULTS - BEST KNOWN MAKESPAN 1582

Optimizer	Makespan					Wilcoxon	Avg Iteration Last Imp	Budget No Imp %	Avg Imp Count	Avg Cts
	Min	Max	Avg	StDev	UB Diff %					
SA-N-E	1668	1887	1737	±49.517	5.44		3346	94.42	225	0.055
DE	1602	1715	1633	±27.302	1.26	-	10007	83.32	23	71.365
ES	1607	1708	1649	±28.614	1.58	-	20482	65.86	23	30.182
GA-N-E-1P	**1586**	1656	1618	±16.387	0.25	-	19393	67.68	27	8.649
GA-N-RI-SBOX	**1586**	1651	1617	±15.246	0.25	-	10945	81.76	26	8.837
PSO	1592	1817	1624	±45.679	0.63	-	31283	47.86	21	11.126
HH-SA-ALL	1634	1734	1664	±27.667	3.29	-	43014	28.31	7	5.871
HH-DE-ES	1591	1625	1608	±9.655	0.57	-	31050	48.25	12	20.506
HH-GA-N-E-SBOX-PSO-SA-N-RI	1620	1666	1622	±10.025	2.4	-	24683	58.86	3	11.522

TABLE III

TAILLARD FSSP 50 JOBS 10 MACHINES – OPTIMIZER RESULTS - BEST KNOWN MAKESPAN 3025

Optimizer	Makespan					Wilcoxon	Avg Iteration Last Imp	Budget No Imp %	Avg Imp Count	Avg Cts
	Min	Max	Avg	StDev	UB Diff %					
SA-N-E	3287	3424	3363	±39.949	8.66		7850	94.77	264	0.137
DE	3110	3226	3164	±29.696	2.81	-	89572	40.29	43	1171.159
ES	3114	3227	3176	±31.661	2.94	-	109775	26.82	45	351.606
GA-N-E-1P	**3067**	3174	3127	±25.001	1.39	-	43033	71.31	47	53.592
GA-N-RI-SBOX	3068	3163	3116	±25.684	1.42	-	52761	64.83	55	55.853
PSO	3100	3610	3163	±104.355	2.48	-	126532	15.65	34	64.363
HH-SA-ALL	3213	3410	3282	±46.285	6.21	-	119170	20.55	9	20.45
HH-DE-ES	3080	3179	3139	±20.544	1.82	-	101850	32.1	15	205.801
HH-GA-N-E-SBOX-PSO-SA-N-RI	3131	3293	3144	±42.09	3.5	-	66763	55.49	3	70.695

TABLE IV

RASTRIGIN 2 DIMENSIONS

Optimizer	Fitness				Wilcoxon	Avg Iteration Last Imp	Budget No Imp %	Avg Imp Count	Avg Cts
	Min	Max	Avg	StDev					
SA-N-E	1.381	63.488	10.708	±12.903		439	92.68	51	0.002
DE	**0.0**	0.005	0.001	±0.001	-	3238	46.03	18	0.259
ES	**0.0**	9.95	1.94	±2.279	-	501	91.65	12	0.016
GA-N-E-1P	**0.0**	6.159	2.954	±2.03	-	1712	71.47	52	0.408
GA-N-RI-SBOX	0.251	34.354	12.748	±9.631	=	30	99.5	2	0.428
PSO	**0.0**	10.278	0.663	±2.256	-	2789	53.52	79	0.141
HH-SA-ALL	0.704	8.148	2.565	±2.128	-	4461	25.65	6	0.077
HH-DE-ES	**0.0**	0.001	0.0	±0.0	-	4959	17.35	16	0.446
HH-GA-N-E-SBOX-PSO-SA-N-RI	**0.0**	3.969	0.728	±1.219	-	2919	51.35	3	0.271

REFERENCES

[1] R. .. .. Vaessens, E. Aarts and J. Lenstra, “Job Shop Scheduling By Local Search,” *Computing Science Notes,* vol. 94/15, 1994.

[2] M. Mastrolilli and O. Svensson, “Improved bounds for flow shop scheduling,” *International Colloquium on Automata, Languages, and Programming,* pp. 677-688, 2009.

[3] D. R. Morrison, S. H. Jacobson, J. Sauppe and E. C. Sewell, “Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning,” *Discrete Optimization,* vol. 19, pp. 79-102, 2016.

[4] I. Poikolainen, F. Neri and F. Caraffini, “Cluster-Based Population Initialization for differential evolution frameworks,” *Information Sciences,* vol. 297, pp. 216-235, 2015.

[5] F. Caraffini, “Stochastic Optimisation Software,” [Online]. Available: https://sites.google.com/site/facaraff/research/sos.

[6] E. Taillard, “Benchmarks For Basic Scheduling Problems,” *European journal of operational research,* vol. 64, no. 2, pp. 278-285, 1989.

[7] D. Wolpert and W. Macready, “No free lunch theorems for optimization,” *IEEE Transactions on Evolutionary Computation ,* vol. 1, no. 1, pp. 67-82, 1997.

[8] GEATbx, “Rastrigin Function,” [Online]. Available: http://www.geatbx.com/docu/fcnindex-01.html#P140_6155.

[9] S. C. D. Kirkpatrick and M. P. Vecchi, “Optimization by simulated annealing,” *Science,* vol. 220, no. 4598, pp. 671-680, 1983.

[10] N. Bhatt and N. R. Chauhan, “Genetic algorithm applications on Job Shop Scheduling Problem: A Review,” in *International Conference on Soft Computing Techniques and Implementations*, Faridabad, 2015.

[11] J. E. Baker, “Reducing bias and inefficiency in the selection algorithm,” *Proceedings of the second international conference on genetic algorithms,* vol. 206, 1987.

[12] J. Kennedy and R. Eberhart, “Particle swarm optimization,” *Proceedings of the IEEE International Conference on Neural Networks,* vol. 4, pp. 1942-1948, 1995.

[13] R. T. Radha, M. Iqbal and K. Umarali, “A particle swarm optimization approach for permutation flow shop scheduling problem,” *International Journal for Simulation and Multidisciplinary Design Optimization ,* vol. 5, 2014.

[14] M. F. Tasgetiren and Y. C. Liang, “A particle swarm optimization algorithm for makespan and total flowtime minimization in the permutation flowshop sequencing problem,” *European journal of operational research,* vol. 177, no. 3, pp. 1930-1947, 2007.

[15] “inspyred: Bio-inspired Algorithms in Python,” [Online]. Available: https://pythonhosted.org/inspyred/.

[16] F. Werner, Genetic Algorithms For Shop Scheduling Problems: A Survey, 2011.

[17] J. H. Drake, A. Kheiri, E. Özcan and E. K. Burke, “Recent advances in selection hyper-heuristics,” *European Journal of Operational Research,* 2019.

[18] C. Stack.AI, “GitHub - Cortical Stack - heuristic optimization platform repository,” [Online]. Available: https://github.com/corticalstack/heuristic-optimization-platform.