A Genetic Programming Based Hyper-heuristic Approach for Combinatorial Optimisation

Su Nguyen
School of Engineering and Computer Science
Victoria University of Wellington
Wellington, New Zealand
su.nguyen@ecs.vuw.ac.nz

Mengjie Zhang
School of Engineering and Computer Science
Victoria University of Wellington
Wellington, New Zealand
mengjie.zhang@ecs.vuw.ac.nz

Mark Johnston
School of Mathematics, Statistics and Operations Research
Victoria University of Wellington
Wellington, New Zealand
mark.johnston@vuw.ac.nz

ABSTRACT

Genetic programming based hyper-heuristics (GPHH) have become popular over the last few years. Most of these proposed GPHH methods have focused on heuristic generation. This study investigates a new application of genetic programming (GP) in the field of hyper-heuristics and proposes a method called GPAM, which employs GP to evolve adaptive mechanisms (AM) to solve hard optimisation problems. The advantage of this method over other heuristic selection methods is the ability of evolved adaptive mechanisms to contain complicated combinations of heuristics and utilise problem solving states for heuristic selection. The method is tested on three problem domains and the results show that GPAM is very competitive when compared with existing hyper-heuristics. An analysis is also provided to gain more understanding of the proposed method.

Categories and Subject Descriptors

I.2 [**Computing Methodologies**]: [Artificial Intelligence]

General Terms

Design, Performance, Algorithms

Keywords

Optimization, Genetic Programming, Hyper-heuristic

1. INTRODUCTION

Hyper-heuristics (HH) have recently emerged as a new search method which explores the heuristic search space instead of the solution space (as heuristics or meta-heuristics do). HHs are described as *"heuristics to choose heuristics"* and aim at *"raising the level of generality"* at which optimisation systems can operate [2]. A key motivating goal for this area is *"the challenge of automating the design and tuning of heuristic methods to solve hard computational search problems"* [5]. Since searching for a good solution in heuristic search space is not trivial and there are no exact methods to find optimal HH solutions, heuristics, meta-heuristics and machine learning methods have naturally become excellent candidates for this task.

Heuristic selection and heuristic generation are currently the two fundamental research directions in HH [5]. Heuristic selection has mainly focused on developing HH frameworks that are able to adaptively select suitable pre-existing heuristics based on the problem solving states and the historical records obtained from the problem solving process. Some successful HH frameworks for heuristic selection such as the choice function [11], [12], Tabu Search based HH [10], Simulated Annealing based HH [15] record the performance of each heuristic as well as the performance of combinations of different heuristics in the heuristic pool. In each iteration, a heuristic is called based on its historical performance and computational time. Some features to diversify HH solutions are also included such as a tabu list of most recent or unsuccessful heuristic calls.

Most heuristic selection methods have used very simple approaches to record or evaluate the performance of each low level heuristic (LLH) or the combination of two LLHs as shown in [11], [12]. However, in order to be effective, hyper-heuristics may require even more complicated combinations of LLHs, for example, the sequential implementation of three or more LLHs. Another limitation of previous studies in heuristic selection is the lack of using problem solving state to enhance the performance of HH. For example, LLHs with more exploitation such as local search could be very powerful in many cases, however, it may be less effective when the current solution is trapped at local optima, where some LLHs with mutation could be more useful.

The objective of heuristic generation methods, on the other hand, is to fabricate a new heuristic (or meta-heuristic). The obtained heuristic can be either an improving or constructive heuristic. In order to generate a new heuristic, the HH framework must be able to combine various small components (normally common statistics or operations used in pre-existing heuristics) and these heuristics are trained on a training set and evolved to become more effective.

Genetic Programming (GP) [18] is an evolutionary computation method aiming to evolve a population of programs, traditionally represented as tree structures. In the last few years, GP has been used very often in the field of hyper-

GECCO'11, July 12–16, 2011, Dublin, Ireland.

heuristics [7]. Most common applications of GP in HH are to automatically generate key components of problem solving methods (heuristics, meta-heuristics). In this paper, GP is used as a heuristic selection method to solve combinatorial optimisation problems. The objectives of this study are:

1. *To employ GP system to incorporate problem solving states and improve the heuristic selection process.* It is expected that the proposed GP, with its flexible representation, can overcome the limitations encountered by other heuristic selection methods.
2. *To compare the performance of the proposed method with other HHs.* Since using GP as a heuristic selection method is quite new, it is interesting to see its performance compared with other HHs for heuristic selection to confirm the feasibility of this method.
3. *To analyse the proposed method to gain more understanding of its behaviours as well as investigate the possibility for further improvements.* The analysis will help explain how the proposed GP system evolves adaptive mechanisms and what factors could influence its performance.

In the next section, a brief literature review of hyper-heuristics is presented, followed by the methodology of the proposed approach in section 3. Section 4 presents the parameters and experimental design, and the results of these experiments are discussed in section 5. In section 6, an analysis of the proposed method is provided. Conclusions and future research are given in section 7.

2. LITERATURE REVIEW

In the last decade, many novel hyper-heuristic methods have been proposed and shown very promising results. In general, hyper-heuristic methods can be classified into two main categories [5]: (1) heuristic selection and (2) heuristic generation.

2.1 Heuristic selection methods

Several new genetic algorithms have been applied to hyper-heuristics. For instance, Krasnogor et al. [19] showed how a simple inheritance mechanism is capable of learning the best local search to use at different stages of the search. An individual is composed of its genetic material and its memetic material. The memetic material specifies the strategy the individual will use to do local search in the vicinity of the solution encoded in its genetic part. This method is applied to solve different combinatorial optimisation problems and showed very promising results. The analysis also indicated that the algorithm can really learn and make good choices of local search approaches. Another interesting method was developed by Ross et al. [22] for solving one-dimensional bin-packing problems. In this method, the task of GA is to choose the problem states and to associate a small set of heuristics with each of them. A chromosome is composed of blocks, and each block contains the five numbers representing a problem state and an integer that indicates which heuristic is associated with this problem state. The experiments showed promising results for a wide range of instances when compared with known heuristics and the optimal solutions.

In a series of papers, Cowling et al. [11], [12], [13] proposed a hyper-heuristic method based on a choice function. The choice function is the weighted sum of heuristic performance, joint performance of pairs of heuristics, and CPU time from the previous time the low-level heuristic was called. This method adaptively ranks the low-level heuristics based on the choice function and the ranks are then used to decide which heuristic to choose in the next call. This method is simple and can be easily applied to new problem domains. An improved version of this approach was proposed in [12]. However, the performance of the choice function method may depend on the performance of LLHs called at the early stage and the myopic characteristic of this method could prevent it from exploring better solutions, especially when a solution is converged to a local optimum.

More recently, Cuesta-Canada et al. [14] employed Ant colony optimisation (ACO) as hyper-heuristic method to solve the 2D bin packing problem. The Hyper-Heuristic Ant System algorithm (HHAS) uses two sets of heuristics for bin and item selection. A heuristic is represented in HHAS as an array of blocks, each of which includes five variables: quantity of items to be placed by the heuristic defined in the block; rotation decision; item order; bin selection; and item selection. The main components of the ACO part of the HHAS are: the pheromone matrix; the update and decay function; and stagnation behaviour. 25 pheromone matrices (5×5 combinations of decisions) are used to represent the transition between two consecutive variables. Based on the transition from quantity, rotation, item order to bin selection, etc., the path of a heuristic is created according to the possible combinations of the variables used. The experiment showed that HHAS outperformed the choice function based HH, which indicates that more complicated approaches can improve the performance of HH. One of the drawbacks of this method is the use of pheromone matrices; when the number of LLHs grows, it may be difficult for HHAS to collect sufficient information to fill in these matrices.

Local search methods have also been used for heuristic selection. Burke et al. [10] proposed a hyper-heuristic framework for timetabling and rostering in which heuristics compete using rules based on the principles of reinforcement learning and a tabu list of heuristics is maintained which prevents certain heuristics from being chosen at certain times during the search. A multi-objective version of this framework was introduced in [3] to solve space allocation and timetabling. The idea of this multi-objective hyper-heuristic approach is to choose a suitable LLH at each iteration for the optimisation of a given individual objective. Downsland et al. [15] improved the tabu search framework in [10] by integrating the simulated annealing acceptance strategy into the heuristic selection mechanism. The improved framework was used to determine shipper sizes for storage and transportation. Again, the myopic characteristic of a local search based HH method may prevent from creating complicated combinations of LLHs.

2.2 Heuristic generation methods

In recent years, GP has been widely applied to hyper-heuristic frameworks and gained its own name as genetic programming based hyper-heuristics (GP-HH)(see [7] for a comprehensive review). With its flexible structure, GP can be used to build either construction or perturbation heuristics. Bolte et al. [1] may be the first that successfully adopted GP to learn new heuristics. The proposed method used standard GP to evolve annealing schedule functions in simulated annealing to solve the quadratic assignment prob-

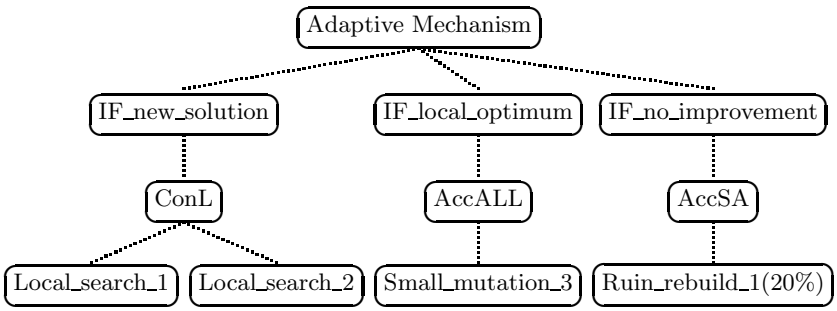


Figure 1: An example program tree for an adaptive mechanism

lem (QAP). Fukunaga [16] proposed CLASS, an automated heuristic discovery system, to develop new local search algorithms for the satisfiability (SAT) problem. This system uses GP to evolve the variable-selection heuristics used in each application of the local search algorithm. The proposed GP evolved new heuristics from a set of primitives that can reformulate any existing heuristic approach in the literature (score, age, rank, conditional branch, etc.). The experimental results showed that the evolved heuristics are very competitive compared with other popular heuristics. Burke et al. [6], [8], [9] evolved new construction heuristics for online bin packing problem with GP-HH. GP generates a priority function of the size of piece p, and the fullness and capacity of bin i. If the output of this function is higher than zero, piece p is placed in bin i. Human designed heuristic can also be found by GP under different trees (individuals).

3. METHODOLOGY

In this study, GP will be used to create an *adaptive mechanism* (AM) to decide which heuristics (or combinations of heuristics) should be performed given a certain state of the problem solving process. In order to achieve this goal, GP must evolve a program that is capable of realising the problem solving states and make effective actions.

3.1 Representation

An advantage of GP for evolving adaptive mechanisms is that GP individuals (represented in tree form) can perform different conditional branching functions; useful for heuristic selection based on the states of the problem solving process. Moreover, good compositions of low level heuristics can also be stored in a subtrees which can be activated by conditional branching functions, and can be evolved by GP. In this paper, GP is used to evolve an adaptive mechanism similar to Iterated Local Search (ILS) [21], i.e., in each iteration some local search heuristics are called to improve the solution followed by different mutation heuristics based on the problem solving states. Three conditional branches are investigated. The subtree of the first branch is called when an initial solution is created or the current solution is mutated or ruined-and-rebuilt. When a local optimal solution is found, the subtree of the second branch is activated. Finally, when some LLHs are performed without any improvement in the best found solution, LLHs in the last branch will be performed.

An example program tree for an adaptive mechanism is shown in Figure 1. In this example, when a new solution is provided, AM will first perform *local search 1* followed by *local search 2*. If a local optimal solution is found, the second branch will be activated and a new solution created by applying *small mutation 3* replaces the current solution. If no improvement is realised in the previous steps, the last branch will be activated to call the *ruin and rebuild 1* to destroy 20% of the current solution and rebuild a new one. However, in this case, the new solution can only replace the current solution if it satisfies the simulated annealing acceptance criteria (*AccSA*). If the branching condition is not satisfied, the subtree of that branch will be ignored. An execution of AM is called an AM iteration. It is noted that several AM iterations need to be performed in order to assess the performance of an AM.

3.2 Function set

Within the subtrees of the three conditional branches, there are two types of functions. The first is the acceptance function which is used to decide whether a solution obtained by a mutation or ruin-and-rebuild heuristic will replace the current solution. Different acceptance functions may create different behaviours of adaptive machanisms. For example, if the all-move (accept all proposed changes) acceptance function is used, it will be easier for AMs to escape from a local optimum but it also reduces the exploitation ability of AMs. On the other hand, if the improve-only acceptance function is used, AMs may quickly trapped at a local optimum. Including these acceptance functions in the function set allows GP to evolve a smarter way to perform mutations.

The second type of function is the heuristic connectors which are used to sequentially execute the left and right subtrees. These connectors allows AMs to represent different combinations of local search heuristics, which help GP overcome the limitation of other HH methods. There are two connectors used in the proposed method: (1) local search connectors to combine local search heuristics in the first branch, and (2) perturbation connectors to combine mutation or ruin-and-rebuild heuristics along with their acceptance functions in the second and the third conditional branch as shown in Figure 1.

3.3 Terminal set

The terminal set includes all the LLHs available in a problem domain and these heuristics are categorised into three groups: local search, mutation, and ruin-and-rebuild heuristics. In the proposed representation, terminals in the local search group can only appear in the first branch of GP tree and terminals of the mutation group and the ruin/rebuild group can only included in the second and the third branch as an argument of acceptance functions.

3.4 Fitness function

There are several ways to measure the performance of an adaptive mechanism such as quality of obtained solutions,

computational time, etc. In this study, we only focus on the quality of the solution obtained by AM. Given a particular solution, the fitness of an adaptive mechanism is evaluated by calculating the objective value of the best solution obtained through a sequence of N applications of that adaptive mechanism. Within each application of AM, a solution is transformed by LLHs included in that AM in the order determined by the tree representation presented in previous section. Basically, this fitness function measures the effectiveness of an adaptive mechanism, which is either the ability to exploit and improve a given solution provided by the combinations of local search heuristics, or the ability to explore the solution search space for new potential solutions given by the combinations of mutation and ruin/rebuild heuristics.

3.5 Proposed algorithm

The role of GP in this study is to choose from a given set of low level heuristics to construct an adaptive mechanism that can provide high quality solutions for a specific problem instance. Note that there are two objects being evolved simultaneously in this study. The first is the adaptive mechanism within a GP population, which is produced by the evolutionary process. The second is the problem solution obtained by applying the evolved adaptive mechanism. The genetic programming method for evolving adaptive mechanisms (GPAM) is presented in Algorithm 1. The method

Algorithm 1: GPAM algorithm

```
ℍ_l, ℍ_m, ℍ_r ← load problem domain heuristics;
𝔻 ← load problem instance;
s ← generate random solution for 𝔻;
s* ← s;  //s* is the best found solution
ℙ ←Initialise population using ℍ_l, ℍ_m, ℍ_r;
for n ← 0 to maxGeneration do
    foreach adaptive mechanism i in ℙ do
        s_i ← the best solution resulting from a sequence
        of N applications of i starting from initial
        solution s;
        fitness_i ← objective value of s_i;
        if s_i is better than s* then
            s* ← s_i;
        end
    end
    ℙ ← apply reproduction, crossover, mutation to ℙ ;
    s ← s*;
end
```

starts by loading all heuristics available for a specific problem domain. These heuristics will serve as terminals in GP. The heuristics can be classified into three subsets $\mathbb{H}_l, \mathbb{H}_m, \mathbb{H}_r$ which contain local search, mutation and ruin-and-rebuild heuristics respectively. The problem instance $\mathbb{D}$ that needs to be solved is then loaded and a single random solution s is generated. The GP population is initialised based on the function set mentioned above and terminal set of $\mathbb{H}_l, \mathbb{H}_m, \mathbb{H}_r$. In each generation, each individual in the population is an adaptive mechanism. N is the number of iterations an adaptive mechanism is executed to find better solutions for $\mathbb{D}$ starting from initial solutions s. N is an important parameter in GPAM because small N may not be sufficient to accurately assess the quality of an adaptive mechanism, while high N may increase computational time of GPAM. The best solution s_i found by each individual is stored and its objective value is used as fitness of that individual. The new population is then created through reproduction, crossover and mutation process. The best found solution s^* found in the previous generation is assigned to the initial solution s and GP starts the new generation. In this algorithm, it is obvious that GPAM is not only a search method to find good solutions for a problem instance but also a heuristic selection method which tries to improve the performance of the adaptive mechanism.

4. EXPERIMENTAL DESIGN

In this study, HyFlex (Hyper-heuristics Flexible framework) [4], a Java framework for the implementation hyper-heuristics, is used to test performance of GPAM. HyFlex has a collection of different problem domains and heuristics used to solve these problems.

4.1 Problem domains

In the experiments, three problem domains available in HyFlex [17] are used: maximum satisfiability problem (MAX-SAT), one dimensional bin packing, and permutation flow shop. For each problem domain, the default problem instances in HyFlex are used to measure the performance of GPAM.

4.1.1 MAX-SAT

Boolean satisfiability (SAT) is a problem of determining if there is an assignment of boolean variables in a formula that can make that formula evaluate to *true*. For example, given a problem of n variables $F(x) = (x_1 \vee x_2) \wedge (x_{n-1} \vee x_3 \vee x_{16}) \wedge ... \wedge (x_2 \vee x_n)$, the task is to find the truth assignment for each x_i for all $i = 1, ..., n$ such that $F(x) = TRUE$. One of the optimisation extensions of SAT is MAX-SAT in which the goal is to find an assignment of the boolean variables that maximise the number of clauses in $F(x)$ that are true. In MAX-SAT, each clause is a disjunction of a set of variables.

4.1.2 Bin packing

Bin packing is a traditional NP-hard problem in combinatorial optimisation. The objective of this problem is to pack a number of given items or pieces using the least number of bins. The main constraint is that the sum of all pieces cannot exceed the capacity of the bin. In order to avoid large plateaus in the search space around best solutions, $fitness = 1 - \frac{1}{n}\sum_{i=1}^{n}(fullness_i/C)^2$ is used as objective value instead of the total number of bins, where $fullness_i$ is the sum of all pieces in bin i, C is the capacity of the bin and n is the number of bins being packed.

4.1.3 Flow-shop scheduling

Flow-shop scheduling is the problem of finding the best order of n jobs to be processed on m consecutive machines in order to minimise the makespan (the completion time of the last job to exit the shop). In permutation flow-shop scheduling, all jobs have to be processed through a pre-determined sequence of machines and no job can jump over any other jobs. A machine can only process a single job and if a job comes to a busy machine, it has to wait in the queue. A machine can be idle only if there are no jobs ready to be processed by it.

4.2 GP parameters

The GP part of GPAM is developed based on ECJ19 [20], an evolutionary computation software package developed at

George Mason University. Since this paper focuses on the idea of applying GP for heuristic selection, only the simple standard GP system is used in order to avoid hindering the analysis of the results. The GP parameters for the experiments are shown in Table 1.

Table 1: Parameters of the proposed GPAM

Population Size	50
Crossover rate	90%
Mutation rate	5%
Reproduction rate	5%
Generations	50
Max-depth	5
Function set	*AM, IfNewSol, IfLocalOpt, IfNoImproveN, AccSA, AccALL, AccIO, ConL, Con*
Terminal set	*LS(m,p), MT(m,p),RUIN(m,p)*
Fitness	objective value of the solution found by AM

The initial GP population is created using ramped-half-and-half [18]. Tournament selection with the size of 7 is used to select individual for genetic operations. Standard single point crossover and point mutation are used to produce new individuals. The function set includes all the functions discussed in section 3 and Table 1, where $AccSA$, $AccALL$, $AccIO$ stand for simulated annealing, all move, and improve-only acceptance function of heuristics; $ConL$ and Con are the connectors of local search and mutation/ruin heuristics respectively; and LS (m,p), MT (m,p), $RUIN$ (m,p) are elements of $\mathbb{H}_l, \mathbb{H}_m, \mathbb{H}_r$ respectively, where parameter m indicates which specific mode of the low level heuristics to be used (e.g. steepest descent local search) and parameter p indicates the strength of a low level heuristic (e.g. the number of elements in the solution to be mutated). Both m and p are treated as ephemeral random constants (ERC). In all experiments, the fitness of an adaptive mechanism is measured by executing it for $N = 10$ iterations, as described in section 3.4.

5. RESULTS

This section presents the performance of GPAM on three problem domains. The performance is also compared with the top two hyper-heuristics (HH_4 and HH_6) within the eight default hyper-heuristics developed by ASAP group at the University of Nottingham [17]. Moreover, a random version of the proposed method (rGPAM) is also implemented. In rGPAM, all terminals are randomly regenerated in each iteration and only the AM structures, which are the combinations of LLHs created by connectors and acceptance functions, are evolved by GP. rGPAM is equivalent to a random iterated local search, in which the local search and mutation parts are randomly selected. Comparison with rGPAM will help explain whether the good solutions are obtained by logically combining LLHs or just by the effectiveness of pre-existing LLHs. In order to make a fair comparison, GPAM and rGPAM always start with the same initial solution and the results in this section are the best values of problem instances obtained by one run of each proposed HH. All of the problem domains are minimisation problems.

5.1 MAX-SAT

Table 2 shows the results obtained by GPAM and other approaches for ten default MAX-SAT instances in HyFlex, which contain between 311 and 744 variables, and between 2200 and 3500 clauses. The objective value is the number of unsatisfied clauses. The best objective value for each instance is highlighting in bold. Table 2 shows that GPAM is a very competitive hyper-heuristic method to solve MAX-SAT problem. GPAM provides a better result than HH_4 and HH_6 in most instances. It is also quite interesting that rGPAM also found very good results in these instances, even better than those obtained by HH_4 and HH_6 in several instances. Since many successful methods to solve MAX-SAT are based on randomness (e.g WalkSAT, Novelty), it is possible that rGPAM with high-diversity population can be better than more systematic HHs. The good performance of GPAM indicates the exploration ability of this method.

Table 2: Objective values of MAX-SAT problems

Instance	GPAM	rGPAM	HH_4	HH_6
#1	**8**	16	65	19
#2	33	38	48	**29**
#3	24	**22**	45	34
#4	**8**	**8**	9	11
#5	6	**5**	16	9
#6	7	**4**	17	7
#7	11	**8**	18	18
#8	**6**	7	37	37
#9	7	11	16	**5**
#10	**6**	7	18	7

Table 3: Objective values of Bin Packing problems

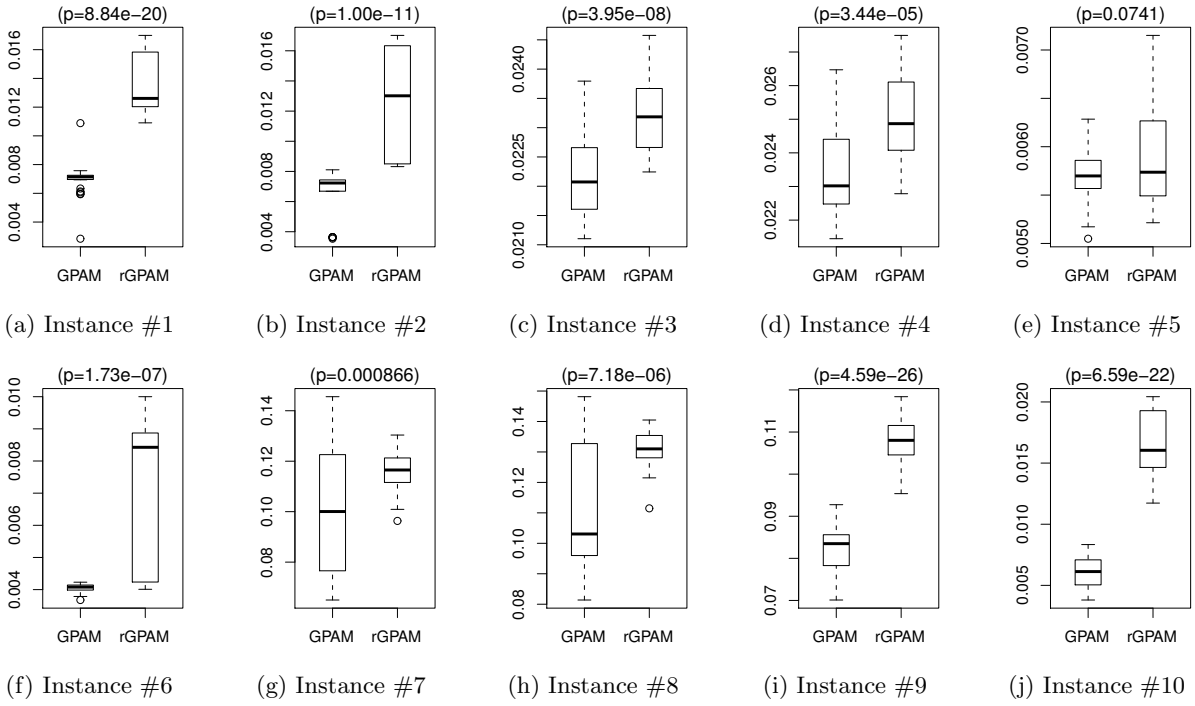
Instance	GPAM	rGPAM	HH_4	HH_6
#1	**0.007165**	0.012618	0.016151	0.016235
#2	**0.007559**	0.016346	0.011968	0.016388
#3	**0.021713**	0.023184	0.022526	0.023137
#4	**0.023108**	0.024418	0.023530	0.023596
#5	0.005604	0.006840	**0.004754**	0.012858
#6	0.003985	0.008428	**0.003146**	0.015671
#7	0.069706	0.105994	**0.026893**	0.094808
#8	0.104935	0.123926	**0.054556**	0.118187
#9	0.070929	0.116033	0.087692	**0.062025**
#10	**0.00503**	0.014882	0.025665	0.017577

5.2 One dimensional bin packing

The experimental results are shown in Table 3. In this problem domain, GPAM shows very promising result. GPAM provides the best results in five problem instances and second-best in the other five. In the two most difficult instances (7 and 8) with a triplet distribution of pieces (a well-filled bin must contain one big item and two small items), GPAM is only slightly dominated by HH_4 but better than HH_6. Different from MAX-SAT problems, rGPAM cannot compete with other HHs methods and it is worse than GPAM in all cases. The results emphasise the important of heuristic learning which allows HHs to smartly select suitable heuristics.

5.3 Permutation flow-shop scheduling

In this experiment, seven hard instances are used to access the performance of GPAM. Since there are more LLHs for this problem domain, *crossover rate* of 80% and *mutation rate* of 20% are used in order to increase the diversity of the GP population. The results for this problem domain are shown in Table 4. The results again show that most solutions obtained by rGPAM are dominated by other HHs. Although GPAM only provides the best solutions for one instance in this problem domain, other solutions found by GPAM are



(a) Instance #1 (b) Instance #2 (c) Instance #3 (d) Instance #4 (e) Instance #5

(f) Instance #6 (g) Instance #7 (h) Instance #8 (i) Instance #9 (j) Instance #10

Figure 2: Performance of GPAM and rGPAM on the bin packing problem (30 runs)

Table 4: Objective values of permutation flow-shop scheduling problems

Instance	GPAM	rGPAM	HH_4	HH_6
#1	6310	6344	**6305**	6314
#2	6270	6284	6276	**6242**
#3	6340	6344	6346	**6337**
#4	**6323**	6369	6353	6332
#5	6421	6429	6422	**6403**
#6	10501	10515	**10497**	**10497**
#7	10944	10941	10957	**10923**

very competitive with HH_4 and HH_6; in 6 out of 7 instances, GPAM is either the best or the second best method. It is also noted the local search heuristics employed in this problem domain mainly differ from one another only in their efforts to explore the neighbourhood of the current solution. In this case, it could be better to consider the computational time to measure the quality of adaptive mechanisms because the behaviors of these local search heuristics are quite similar.

6. FURTHER ANALYSIS

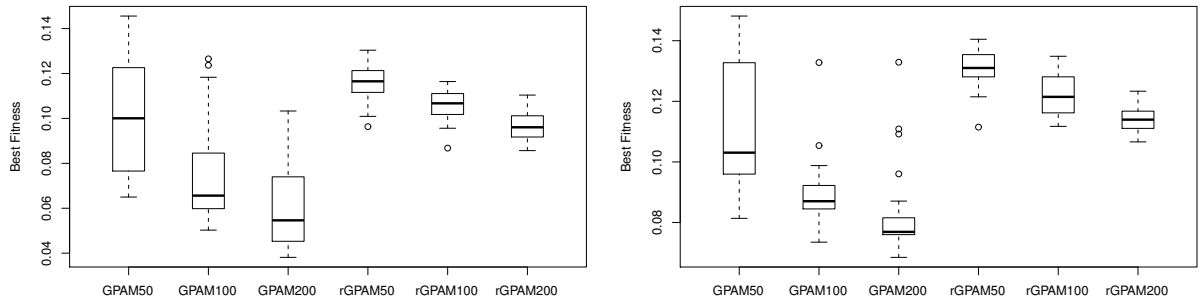
Previous experiments show that GPAM performs well on different problem domains. In this section, an analysis of GPAM is given in order to gain better understanding of this method. The Bin Packing problem is used here as an example because this problem domain includes a good collection of low level heuristics and a wide range of problem instances which are very useful for the analysis.

6.1 GPAM vs. rGPAM

Thirty independent runs of GPAM and rGPAM are used to check the statistical significance. It is also noted that the same initial solution of a problem instance is used in the thirty runs to avoid the biased results caused by random initial solutions.

Figure 2 shows the best fitness obtained by GPAM and rGPAM on the ten Bin Packing instances. GPAM significantly outperforms rGPAM in all instances (T-test, $p \ll 0.05$) except for instance #5. It is also more robust than rGPAM in most instances except for instance 7 and 8, which are the two most difficult instances where pieces follow the triplet distribution; in these instances, GPAM has wider range of solutions. Although the best solutions are found by GPAM, it is also the one that results in the worst solutions. One explanation for these behaviours is that these instances are more difficult than the others and require different combinations of low level heuristics at different states of problem solving process. Since GPAM in this study only uses the population size of 50, it is possible that the population cannot preserve the diversity at the latter stage of the evolution process. rGPAM, on the other hand, is always capable of executing all low level heuristics. This property makes rGPAM more robust than GPAM in the two cases.

In order to confirm whether the behaviours observed from instance 7 and 8 of the Bin Packing problem is because of the lack of diversity in the population, GPAM with larger population sizes is tested to solve instance 7 and 8. The results for these experiments are presented in Figure 3 where three values of the population size (50, 100, 200) are tested. The labels in the x-axis is the name of the method and the population size used to solve the problem instance (e.g. rGPAM100 shows the peformance of rGPAM with the population size of 100). For instance 7, the population size influences GPAM more strongly than rGPAM. The average (best) fitness of GPAM decreases from 0.99945 with *population size* 50 to 0.06078 with the *population size* 200, while rGPAM only decreases from 0.116389 to 0.096118 with the same increase in the population size. The standard deviation of GPAM is also reduced but still larger than that of rGPAM. In case of instance 8, both average and standard deviation of best



(a) Bin Packing - Instance #7

(b) Bin Packing - Instance #8

Figure 3: Influence of the population size on the performance of GPAM and rGPAM

fitness are improved significantly when the population size increase from 50 to 200, except for a few outliers. These experiments support the assumption that it is possible to improve the performance of GPAM by increasing the diversity in the GPAM population.

6.2 GPAM behaviours

The rest of this analysis will focus mainly on the behaviours of GPAM within a specific run. Figure 4 shows the performance of GPAM for instance 1 of the bin packing problem. One interesting point in GPAM is that the fitness of the GP population converges very fast to the best fitness because the initial solution to measure the quality of evolved adaptive mechanisms is updated by the best found solution after each generation. Different from ordinary evolutionary algorithms, it is noted that the convergence of fitness is not equivalent to the convergence of the population. This property allows GPAM to preserve the diversity of AMs in the population, which help GP continue to improve the problem solution.

The average fitness of GPAM with various population sizes and number of AM iterations N is presented in Figure 5 and Figure 6. These parameters are important because it strongly influences the computation time of GPAM. It is noticed that GPAM with larger population size tends to converge faster but the behaviour is not much different when population size is more than 150. The number of AM iterations N shows a strong influence on the performance of GPAM when it is increased from 5 to 40. It can be observed that the increase of N and population size do help GPAM find better solutions. While the large population size improves GPAM by creating the diversity in the population, the large N helps GPAM assess the quality of AMs more accurately to make better selection decisions.

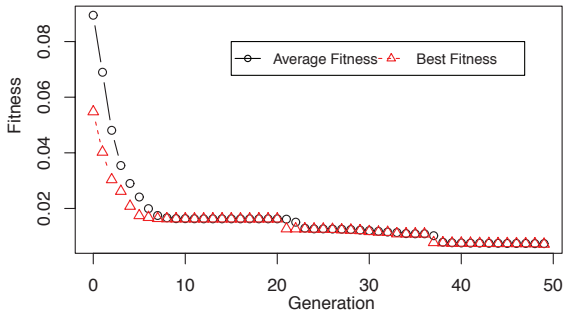


Figure 4: Performance of GPAM on instance #1 of the bin packing problem

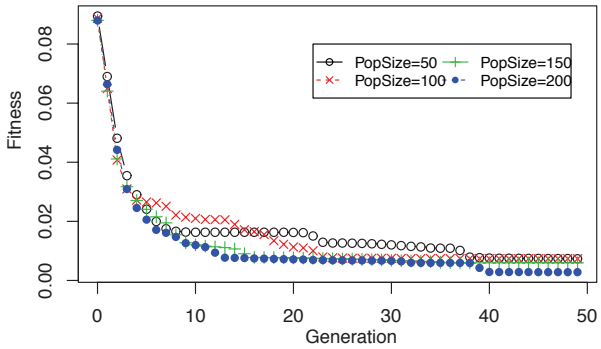


Figure 5: Performance of GPAM with different population sizes on instance #1 the bin packing problem ($N = 10$)

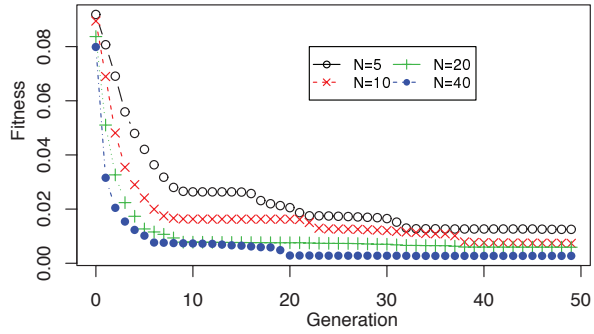


Figure 6: Performance of GPAM with different values of N on instance #1 of the bin packing problem ($Popsize = 50$)

The average and standard deviation of best fitness of 30 runs of GPAM with different combinations of population size and N are presented in Table 5. It is noted that GPAM requires more computational effort to obtain the result in the lower right of Table 5. The largest improvements are observed along the diagonal of Table 5 although lesser improvements can also be realised when moving to the right or down the table. These observations suggest that the computational effort should be shared between the population size and N in order to achieve the greatest improvement. The values of the standard deviation in Table 5 do not show a clear improvement except for the one at the botom right with the *population size* 200 and $N = 40$. The results of GPAM with a population size of 200 and $N = 40$ for other instances have also been significantly improved (not presented here due to page limit).

Table 5: Performance of different combinations of the population size and N for instance #1 (30 runs)

Mean ± Std		N		
		10	20	40
Population size	50	0.006979 ± 0.001143	0.006316 ± 0.001794	0.005875 ± 0.001656
	100	0.006535 ± 0.001098	0.004577 ± 0.001817	0.004276 ± 0.001736
	200	0.005890 ± 0.001600	0.004329 ± 0.001799	0.003123 ± 0.000958

7. CONCLUSIONS

This paper proposed GPAM, a new hyper-heuristic method based on genetic programming. The novelty of this method is that GP is used for heuristic selection instead of heuristic generation like previous genetic programming based hyper-heuristics. The representation of GPAM allows GP to evolve more complicated combinations of heuristics and incorporate problem solving states to support the selection of heuristics. The proposed method have been tested on three problem domains and shown very promising results when compared to existing hyper-heuristics and the random heuristic selection method. The experimental results indicate that GPAM is a robust HH method which provides good quality solutions for different problem domains. Moreover, these results have shown that systematic heuristic selection methods can solve problems more effectively than the random heuristic selection method. An analysis of GPAM has also been given to gain better understanding of the proposed method. The analysis suggests that maintaining diversity in the population is important, especially when dealing with difficult problems. In addition, the computational effort should be reasonably shared between the population size and the number of AM iterations.

In future studies, a more extensive analysis of GPAM will be investigated to improve its performance. An analysis of evolved adaptive mechanisms would be important to explain the behaviours of GPAM. It would also be interesting to try other representations and fitness functions and study how they influence the performance of GPAM.

8. REFERENCES

[1] A. Bolte and U. W. Thonemann. Optimizing simulated annealing schedules with genetic programming. *European Journal of Operational Research*, 92(2):402–416, 1996.

[2] E. Burke, G. Kendall, J. Newall, E. Hart, P. Ross, and S. Schulenburg. Hyper-heuristics: An emerging direction in modern search technology. In *Handbook of Metaheuristics*, pages 457–474, 2003.

[3] E. Burke, J. Landa Silva, and S. E. Multi-objective hyper-heuristic approaches for space allocation and timetabling. *Meta-heuristics: Progress as Real Problem Solvers*, pages 129–158, 2005.

[4] E. K. Burke, T. Curtois, M. R. Hyde, G. Kendall, G. Ochoa, S. Petrovic, J. A. V. Rodríguez, and M. Gendreau. Iterated local search vs. hyper-heuristics: Towards general-purpose search algorithms. In *IEEE CEC'10: Congress on Evolutionary Computation*, pages 1–8, 2010.

[5] E. K. Burke, M. Hyde, G. Kendall, G. Ochoa, E. Ozcan, and R. Qu. Hyper-heuristics: A survey of the state of the art. Technical Report Computer Science Technical Report No. NOTTCS-TR-SUB-0906241418-2747, School of Computer Science and Information Technology, University of Nottingham, 2010.

[6] E. K. Burke, M. R. Hyde, and G. Kendall. Evolving bin packing heuristics with genetic programming. In *PPSN*, pages 860–869, 2006.

[7] E. K. Burke, M. R. Hyde, G. Kendall, G. Ochoa, E. Ozcan, and J. R. Woodward. Exploring hyper-heuristic methodologies with genetic programming. *Artificial Evolution*, 1:177–201, 2009.

[8] E. K. Burke, M. R. Hyde, G. Kendall, and J. Woodward. Automatic heuristic generation with genetic programming: evolving a jack-of-all-trades or a master of one. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1559–1565, 2007.

[9] E. K. Burke, M. R. Hyde, G. Kendall, and J. R. Woodward. The scalability of evolved online bin packing heuristics. In *IEEE CEC '07*, pages 2530–2537, 2007.

[10] E. K. Burke, G. Kendall, and E. Soubeiga. A tabu-search hyperheuristic for timetabling and rostering. *Journal of Heuristics*, 9(6):451–470, 2003.

[11] P. Cowling, G. Kendall, and E. Soubeiga. A hyperheuristic approach to scheduling a sales summit. In *PATAT '00: Selected papers from the Third International Conference on Practice and Theory of Automated Timetabling III*, pages 176–190, 2000.

[12] P. Cowling, G. Kendall, and E. Soubeiga. A parameter-free hyperheuristic for scheduling a sales summit. In *Proceedings of the 4th Metaheuristic International Conference, MIC 2001*, pages 127–131, 2001.

[13] P. Cowling, G. Kendall, and E. Soubeiga. Hyperheuristics: A tool for rapid prototyping in scheduling and optimisation. In *EvoWorkShops. LNCS*, pages 1–10. Springer, 2002.

[14] A. Cuesta-Canada, L. Garrido, and H. Terashima-Marin. Building hyper-heuristics through ant colony optimization for the 2D bin packing problem. In *Knowledge-Based Intelligent Information and Engineering Systems*, LNCS 3684, pages 654–660, 2005.

[15] K. A. Dowsland, E. Soubeiga, and E. Burke. A simulated annealing based hyperheuristic for determining shipper sizes for storage and transportation. *European Journal of Operational Research*, 179(3):759–774, 2007.

[16] A. Fukunaga. Automated discovery of composite sat variable-selection heuristics. In *Eighteenth national conference on Artificial intelligence*, pages 641–648, 2002.

[17] M. Hyde and G. Ochoa. Cross-domain heuristic search challenge (CHeSC), http://www.asap.cs.nott.ac.uk/chesc2011/.

[18] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[19] N. Krasnogor and J. Smith. Emergence of profitable search strategies based on a simple inheritance mechanism. In *GECCO '01*, pages 432–439, 2001.

[20] S. Luke. *Essentials of Metaheuristics*. Lulu, 2009. available at http://cs.gmu.edu/∼sean/book/ metaheuristics/.

[21] H. Ramalhinho-Lourenco, O. C. Martin, and T. Stutzle. Iterated local search. In *Handbook of Metaheuristics*, International Series in Operations Research & Management Science, pages 321–353, 2003.

[22] P. Ross, J. G. Marín-Blázquez, S. Schulenburg, and E. Hart. Learning a procedure that can solve hard bin-packing problems: a new ga-based approach to hyper-heuristics. In *GECCO '03*, pages 1295–1306, 2003.

[23] J. Woodward, A. Parkes, and G. Ochoa. A mathematical formalization of hyper-heuristics. workshop on hyper-heuristics automating the heuristic design process. In *10th International Conference on Parallel Problem Solving From Nature (PPSN-08)*, 2008.