Automatic Configuration of Metaheuristic Algorithms for Complex Combinatorial Optimization Problems

Yiliang Xu, Meng Hiot Lim, Yew-Soon Ong

Abstract—We report our work on the algorithmic development of an evolutionary methodology for automatic configuration of metaheuristic algorithms for solving complex combinatorial optimization problems. We term it Automatic Configuration Engine for Metaheuristics (ACEM). We first propose a novel Left Variation - Right Property (LVRP) tree structure to manage various metaheuristic procedures and properties. With LVRP tree, feasible configurations of metaheuristics can be easily specified. An evolutionary learning algorithm is then proposed to evolve the internal context of the trees based on pre-selected training set. Guided by a user-defined satisfaction function of the candidate algorithms, it converges to the optimal or a very good algorithm. The experimental comparison with two recent state-of-the-art algorithms for solving the quadratic assignment problem (QAP) shows that ACEM produces an hybrid-genetic algorithm with human-competitive or even better performance.

I. INTRODUCTION

COMBINATORIAL optimization (CO) problems arise in a large number of real-life applications. Classic examples include the quadratic assignment problem (QAP) [1] and the travelling salesman problem (TSP) [2]. Many CO problems are NP-hard. With the conjecture P≠NP, exact algorithms do not exist. Even worse, approximation algorithms with guaranteed approximation bounds have proven not to exist for some CO problems (e.g., TSP without triangle inequality). As a result, many heuristic or more recently, metaheuristic based approaches have surfaced to address the need for high-quality solutions in reasonable time. Examples of metaheuristics include genetic algorithm [3], ant colony optimization [4], tabu search [5], [6] and more recently, memetic algorithm [7].

Based on the No Free Lunch (NFL) theorem [8], there does not exist a universal black-box algorithm, such as metaheuristic, which performs “well” for all the optimization problems. For particular problem domain, one has to configure highly specialized algorithms for “good” solution. However, in the conventional algorithm development, the performance of the highly specialized metaheuristic algorithm depends very much on the designer’s expertise on both algorithm design and the knowledge specific to the problem domain. So far, there have been few attempts to address the need for automatic optimization of the configurations of metaheuristic algorithms for solving a particular class of problems.

With these considerations in mind, a novel evolutionary method is proposed to optimize the configurations of the metaheuristic algorithms. We call it Automatic Configuration Engine for Metaheuristics (ACEM). Users or system integrators with modest background knowledge and expertise on metaheuristic algorithms design may use ACEM to automatically search for optimal or very good configurations of metaheuristics for solving a class of CO problems.

The rest of the paper is organized as follows. In Section II, we present the related work. Section III defines the problem of automatic configuration of metaheuristics for CO problems. The algorithm of ACEM is presented in Section IV. Experimental results on Quadratic Assignment Problem (QAP) benchmarks are presented in Section V before we conclude the paper in Section VI.

II. RELATED WORK

In recent years, researchers began to pay great attention in developing technologies to automate the algorithm design process so as to reduce the reliance on human’s experience and inference. Most of these studies are still in the infancy stage. There are even fewer attempts to address the need for optimization of various metaheuristic algorithms as ACEM in this paper. Some representative related works are reviewed in this section.

The most well-known relevant work is called Genetic Programming (GP) [9]. GP usually encodes solution methods or systems in tree based artificial chromosomes. It adopts the operations in genetic algorithm, such as crossover and mutation, to evolve the chromosomes. GP optimizes not only the parameter settings, but also the structures of the solution method for a given problem. GP successfully produces many human-competitive results in a wide range of real-world applications. However so far, it is only applicable to relatively simple systems, such as function approximation, RLC circuits design or even a programming problem as in [10]. It cannot address the need for automatic generation of complex metaheuristic algorithms for general optimization as required in many real-world applications.

In Birattari’s Ph.D. thesis [11], a series of algorithms, namely F-race algorithms, for tuning metaheuristics are presented. Recent improvements on the F-race algorithm can be found in [12]. The basic idea of the F-race algorithms is to abstract N tunable features (e.g. real number parameters, choice of procedures etc.) and formulate the tuning problem as an N-dimensional optimization problem. This
optimization problem is solved by a racing algorithm [13], which works by evaluating the candidate configurations and meanwhile eliminating the inferior candidate configuration instances or branches based on a statistical test. This way, the search space is narrowed down dramatically along with the progress of the tuning. However, when the search space is highly dimensional, the complexity of the F-race search could be impractically expensive. This makes the F-race algorithms only applicable with very limited dimensions. In contrast, ACEM works in a very different manner. It formulates the search space using a novel tree structure and adopts a self-adaptive learning mechanism to converge the search to a very good configuration. This way, current implementation of ACEM is able to handle tuning applications with more than 100 dimensions. As a result, F-Race algorithm produces algorithms that only solve relatively simple and low-dimensional QAP benchmark problems while ACEM produces algorithms with human-competitive performance on very large QAP benchmarks.

In [14], a linear genetic programming (LGP) approach is proposed to evolve evolutionary algorithms. The basic idea is to represent candidate evolutionary algorithms as sequences of statements in the C programming language. Then LGP technique is applied to evolve the sequences of C statements. A similar work using multi-expression programming (MEP) to evolve evolutionary algorithms can be found in [15]. Compared with AECM, this class of approaches is constrained to standard evolutionary operations like crossover, mutation etc. It does not address the automatic configuration for various metaheuristics. Encoding the evolutionary algorithms into sequences of C codes is also non-trivial. Usually, specialized knowledge and expertise are required. The authors evolved an evolutionary algorithm for solving the quadratic assignment problem (QAP). The simulation results on benchmark instances show no match to that of the resulting algorithm generated by ACEM, which will be presented later in Section V.

Most recently, an evolutionary algorithm (EA) is proposed to evolve evolutionary algorithms [16]. The basic idea is to encode the standard EA’s parameters and operators into a higher-level chromosome individual and apply EA on the higher-level chromosomes. It needs to consider issues such as mis-matching between EA operators. Again, abstracting the tunable “features” and encoding features into chromosome representation is not trivial or generic. Specialized expertise and knowledge on EA are required.

III. PROBLEM DEFINITION

In this section, we define the problem of automatic configuration of metaheuristics for solving CO problems. We begin with the necessary assumptions. Definitions of the input and output are then presented. Based on the assumptions and input/output, the automatic configuration problem is formulated as an constrained optimization problem.

A. Assumptions

We assume the metaheuristic algorithms are configured for solving a class of CO problems. Although we implement the current version of ACEM for solving the class of CO problems with permutation-based solution structure (e.g. QAP and TSP), the methodology is generic to extend to any other string-based solution structure. We assume users can select instances from the problem set as the training set, which represents the class of problems. We also assume that the users are able to evaluate an algorithm \( \text{algo} \) in term of a satisfaction function \( \xi(\text{algo}) \). It embodies the users’ requirements and expectation for the resulting algorithm based on the problem domain-specific information.

B. input and output

The input of the automatic configuration problem is a set of CO problems and the user-defined satisfaction function \( \xi(\cdot) \). The output is the configuration \( \Psi \) of a metaheuristic algorithm \( \text{algo}(\Psi) \). A complete configuration \( \Psi \) consists of,

\[
\begin{align*}
\alpha & \in X: \text{choice of metaheuristic structure}; \\
\beta & \in Y: \text{choice of procedure}; \\
\gamma & \in Z: \text{real number parameters}.
\end{align*}
\]

Therefore \( \Gamma = X \times Y \times Z \) forms the search space.

C. problem formulation

We formulate the automatic configuration problem as a constrained optimization problem as follows,

\[
\Psi^* = \arg \max_{\Psi \in \Gamma} \xi(\text{algo}(\Psi)) \\
\text{s.t. } \text{algo}(\Psi) \text{ is a feasible algorithm.} \quad (1)
\]

IV. ALGORITHM

The optimization problem in Eq. 1 is highly dimensional and thus non-trivial. In this section, we present an Automatic Configuration Engine for Metaheuristics (ACEM) to solve the optimization problem in Eq. 1. We start with the representation of candidate metaheuristics. A novel tree structure, namely left variation-right property (LVRP) tree, is proposed to manage and store various metaheuristic procedures, routines and properties in a systematical way. LVRP tree also enables easy specification of the configurations. An evolutionary learning algorithm is then introduced. Guided by the user defined satisfaction function \( \xi(\cdot) \) and a training set, it evolves the internal context of the LVRP trees to converge to the optimal or a very good configuration of the metaheuristic for solving the class of problems being addressed.

A. Algorithm Modules

A representation for candidate algorithms is required to design and specify metaheuristics. In this work, a flowchart-like structure is chosen as a means of representing an algorithm. We provide algorithm modules as the search space \( X \) in Eq. 1. They are essentially the general frameworks of various classes of algorithms such as genetic algorithms.

Fig. 1 shows some possible flowcharts of genetic algorithm (GA) and hybrid genetic algorithm (hybrid-GA). From the
In the figure, there are up to six components in the GA modules, in particular, Population Initialization, Population Evaluation, Termination Condition, Offspring Producer, Individual Operation and Population Update. First, the Population Initialization procedure produces an initial population of chromosomes. Then, in the generation loop, these chromosomes are evaluated based on the fitness function. In most cases, the statistical information on the population needs to be obtained through the Population Evaluation procedure. After that, the population of individuals is designated as the parent generation. Using the Offspring Producer procedures such as crossover and cloning, the parents reproduce to create members of the offspring generation. Although not necessary in GA, Individual Operations such as mutation and local search are often applied to alter or improve the genetic composition of these existing members of the offspring generation. It is worth noting that in this figure, the dotted arrow indicates an optional flow of the genetic algorithm. In particular, it indicates the conventional genetic algorithm without the augment of Individual Operation such as mutation and local search. In contrast, the solid line indicates the flow of hybrid genetic algorithm which is essentially a conventional GA augmented by one or several Individual Operations. Finally, following the Population Update mechanism, the offspring generation together with the parent generation constitutes a new population for further reproduction. This process is repeated until the Termination Condition is met.

Fig. 2. ACO/hybrid-ACO algorithm modules

B. LVRP Tree

Each component of the modules above is associated to a set of properties and attributes with options for tuning and testing the various configurations of an algorithm. In this work, we propose a novel Left Variation - Right Property (LVRP) tree to manage and organize these properties and attributes, which essentially define the search space $Y \times Z$ in Eq. 1.

There are two categories of nodes in the tree. One is called “real” node and the other is called “virtual” node. Any “real” node represents a “substantial” feature/attribute (like crossover and mutation etc.) of the algorithm. It is denoted by a solid line rectangle, ellipse, hexagon or star. In particular, a solid line rectangle represents a real number parameter; a solid line rectangle, as a root node, represents a component in either GA or ACO module as shown in Fig. 1 and Fig. 2; a solid line ellipse represents a “substantial” feature/attribute (other than a real number parameter) in this component.

Each “real” node has exactly two children. One is called “variation” and the other is called “property” node. Any “real” node represents a “substantial” feature/attribute (like crossover and mutation etc.) of the algorithm. It is denoted by a solid line rectangle, ellipse, hexagon or star. In particular, a solid line hexagon represents a real number parameter; a solid line rectangle, as a root node, represents a component in either GA or ACO module as shown in Fig. 1 and Fig. 2; a solid line ellipse represents a “substantial” feature/attribute (other than a real number parameter) in this component.

“Each “real” node has exactly two children. One is “vari-ations” as the left child and the other is “properties” as the right child. These two nodes are called “virtual” nodes which are denoted by dotted line ellipse and rectangle respectively. They don’t represent any “substantial” feature/attribute of the algorithm but help organize the “real” nodes in a systematic way.

It is worth noting that either “variation” or “property” node can have arbitrary number of “real” children. However, if a “virtual” node (either “variation” or “property”) has no
child, which mean no inherent variation version, sub-routine or parameter, the “virtual” node will be omitted accordingly. It is also worth noting that there is no significant distinguish between “left” and “right”. It is just used for uniformity.

Fig. 3 shows the LVRP tree structure for the Offspring Producer component in Fig. 1. It is used to generate the offspring from the parent population. The left “variation” node of the Offspring Producer in a dashed line ellipse serves as the root of the left subtree to denote the variations of Offspring Producer procedures, such as Crossover, Cloning, etc. The dashed line rectangular block “properties” serves as the root of the right subtree to indicate the properties common to all the variations of Offspring Producer, such as Parent(s) Selection. Similarly, subsequent subtrees are managed and organized in similar structure. For example, Crossover procedures have variations such as Uniform Crossover, Order-1 Crossover, One-point Crossover, which are located in its left subtree. Meanwhile, all these variations have one common feature: Crossover Rate, a real number parameter which is located in the right subtree of Crossover.

The description above shows that this hierarchical tree structure is open to accommodate novel procedures or ideas for embellishment. The new procedures can be located as the siblings of the existing procedures. For example, other well-known or novel crossover operators, such as Partially Matched Crossover (PMX) [3], can be easily introduced to the tree structure by locating them as the siblings of the existing three operators. Their corresponding attributes and features are managed by the left subtree of Crossover. Therefore this structure allows for a system that is open and easily scalable.

C. Configuration Specification

The description earlier shows that the possible “features” associated to an algorithm are located as the leaf nodes in the corresponding LVRP trees. This means that the task of specifying a certain “feature” of an algorithm is essentially to find a path from the root node to a particular leaf node. Therefore configuring a complete algorithm becomes a process of traversing the LVRP trees to find a set of paths starting from the root nodes to the leaf nodes. The configuration of the algorithm is carried out along with the traversal of the tree(s). Therefore it is very natural to think of using the depth-first traversal in this work. To achieve the depth first traversal specific for LVRP trees, a LVRP traversal rule is defined as shown in Fig. 4. The basic idea is to traverse only one child of a “variation” node but every child of a “property” node. This rule is recursively applied to every “real” node in the tree.

Fig. 5 illustrates the flow of a sample configuration of the component Offspring Producer. Starting from the root, it first traverses the variation (left) subtree of Offspring Producer. According to the LVRP traversal rule, only one variation - Crossover, is chosen. It then continues to traverse the corresponding subtrees of (Crossover). Based on the LVRP rule, it traverses the variation (left) subtree of Crossover first. Assuming the user chooses the Uniform Crossover, it reaches the first leaf node, and continues to traverse the property (right) subtree of Crossover. Associated to it is a real number parameter Crossover Rate. By specifying the real value parameter for Crossover Rate, it reaches the second leaf node and goes on to traverse the property (right) subtree of Offspring Producer. From there on, only the Parent(s) Selection property remains to be configured. To do this, it starts by traversing the variation (left) subtree of the Parent(s) Selection. Assuming the user chooses Tournament out of the three variations of Parent(s) Selection available, it reaches the third leaf node. Since the property (right) subtree of Parent(s) Selection is empty, the traversal for this tree as well as the configuration of this component is completed. The traversal covers three paths from the root node to three leaf nodes, Uniform, Crossover Rate and Tournament, respectively. This way, it configures a feasible Offspring Producer instance, which is a uniform crossover with tournament parent selection scheme and a real number crossover rate. The configuration of other components of the algorithm is analogously carried out to specify a complete algorithm. More importantly, based on the LVRP tree structure described earlier, this traversal always yields feasible metaheuristic algorithm, which satisfies the constraint in Eq. 1.
an approach is unmanageable. Therefore a self-adaptive evolutionary learning scheme is developed to address the need for quick and automatic configuration of the metaheuristics.

D. Self-adaptive Evolutionary Learning

As shown in Fig. 5, configuring a metaheuristic algorithm is essentially equivalent to traverse the LVRP trees, finding root-leaf paths. In the progress of finding such paths, the most important step is to make the decision on which branch to choose at each Traversal Split Node (TSN), which is defined as follows,

**Definition 1:** Two categories of nodes are defined as Traversal Split Node (TSN):

1. “variation” node;
2. real number parameter.

It is clear that “variation” nodes belong to the TSN. A decision in choosing a variation of the parent node is required at the TSN. Furthermore, we can assign a number of predefined candidate values to the real number parameters (discretization is applied to avoid infinite search space). Therefore the real number parameter nodes also belong to the TSN. Since every sub-tree of a “property” node needs to be traversed, it does not belong to the TSN.

A trace value is assigned on each branch of a TSN to guide the decision-making. The trace can be viewed as the weight of each branch. Given there are \( m \) number of branches stemming from a TSN and \( \tau^j_i (i = 1, 2, \ldots; j = 1, 2, \ldots, m) \) indicates the trace value of the \( j \)-th branch at the \( i \)-th iteration of the learning. The decision of choosing a branch is based on the trace values,

\[
\text{Choice} = \text{DecisionMake}(\tau^1_i, \tau^2_i, \ldots, \tau^m_i).
\]

The function \( \text{DecisionMake}(\cdot) \) is normally a probabilistic function and it can have many variants. One possible form is based on the simple roulette wheel selection,

\[
P^j_i = \frac{\tau^j_i}{\sum_{j=1}^{m} \tau^j_i},
\]

where \( P^j_i \) indicates the probability of the \( j \)-th branch being chosen at the \( i \)-th iteration. This way, configuring an algorithm through traversing the trees is probabilistically guided by the strength of the trace values. This inspires us to evolve the trace values so that they converge to the optimal or a very good algorithm configuration.

To begin, users first select some sample problem instances as the training set from the class of problems being addressed. The users may specify the expected performance of the algorithm by the satisfaction function \( \xi(\cdot) \), which is gauged by different performance metrics, such as efficiency, effectiveness, robustness, stability, etc. The learning algorithm has the capability to learn from the training set as well as users’ predefined requirements. It adapts the internal context of the LVRP trees iteratively and train them to produce algorithms with progressively better performance.

Fig. 6 shows the flow of the training process of the automatic configuration engine. First, the traces in the trees are initialized. Each branch of a TSN (highlighted in bold) is assigned with an even trace value as shown in Fig. 7. This means every possible configuration setting is awarded the same chance of being selected initially. Then in the evolutionary loop, it configures \( \text{PopSize} \), the number of independent algorithms by traversing the LVRP trees guided by the trace values. The resulting algorithms are evaluated by carrying out test runs on the training set gauged by the user-defined satisfaction function \( \xi(\cdot) \). After that, the trace values in the tree are updated based on the best-performing algorithms in the current iteration. The updated trace values serve for the algorithm configuration in the next iteration. The above steps (except the trace initialization) are repeated until the termination condition is met. Finally the tree branches with the maximum trace values at each TSN are output as the result of the training process.

![Fig. 6. Automatic configuration engine - training phase](image)

Fig. 8 illustrates the process of trace update on \( \text{Offspring Producer} \) as an example. Given that the algorithm configures the best-performing algorithm \( \text{algo}_{i+1} \) based on the trace information of the \( i \)-th iteration, \( \text{algo}_{i+1} \) contains the configurations of \( \text{Offspring Producer} \) as shown in terms of the three bold root-leaf paths in Fig. 8. Then the corresponding trace values along with these three paths are emphasized. A more intuitive example of implementing this update will be illustrated later in Section V. As a result, the relative weight of the trace values along with these three paths are increasing and those of the rest trace values are decreasing. This way, good configuration features are rewarded with a higher chance of being selected in the next iteration and this preference is accumulated in a probabilistic and statistical manner as the learning progresses. Finally, it converges with a few root-leaf paths dominating the trees by large relative trace values along with them. These paths, i.e. the
configurations, constitute a good algorithm derived from the training. It needs to be verified by carrying out test runs on the set of test problems.

It is necessary to mention that ACEM has the capacity to learn from not only the specific nature of the problem being addressed, but also the user requirements on the algorithm, which is reflected by the satisfaction function. The domain-specific nature of the problem, together with the user requirements, guides the search in converging toward configurations suitable for the real-life scenarios.

V. EXPERIMENTS

To validate the usefulness and effectiveness of ACEM, we test it on a diverse set of benchmark instances of a well-known CO problem, namely the quadratic assignment problem (QAP). With two different user-defined satisfaction functions, two hybrid genetic algorithms are derived from the training, each satisfying users’ requirements.

A. Quadratic Assignment Problem

The quadratic assignment problem (QAP) was first introduced by Koopmans and Beckmann [1] to solve a facility location problem. It is formulated as follows. A problem of size \( n \), can be represented by two \((n \times n)\) matrices \( A = [a_{ij}] \) and \( B = [b_{ij}] \). Matrix \( A \) can be interpreted as a distance matrix, i.e., \( a_{ij} \) represents the distance between location \( i \) and \( j \). \( B \) is normally referred to as the flow matrix, i.e., \( b_{ij} \) denotes the flow of materials from facility \( i \) to \( j \). The goal is to find a permutation \( \pi \) of the set \( M = \{1, 2, ..., n\} \), to minimize the cost function,

\[
C(\pi) = \sum_{l=1}^{n} \sum_{t=1}^{n} a_{lt}b_{\pi(l)\pi(t)}. \tag{3}
\]

where \( \pi(l) \) and \( \pi(t) \) denote the \( l \)-th and the \( t \)-th element in the permutation \( \pi \), respectively.

QAP is NP-hard. Problem instances with size greater than 25 are generally considered as unsolvable. QAP has found many applications [17] in real-life. Due to its applicable and theoretical importance as well as its computational complexity, many metaheuristic algorithms have been proposed to solve the QAP. Examples include genetic algorithm (GA) [18]–[20], ant colony optimization (ACO) [4], [21], simulated annealing [22], tabu search [23], iterated local search (ILS) [24] and so on. A recent survey of QAP can be found in [17]. The analysis of various metaheuristic algorithms on QAP can be found in [26].

B. Algorithm Evaluation

Before we start, it is necessary to consider how we evaluate the performance of an algorithm. This essentially relates to the formulation of the satisfaction function \( \xi(\cdot) \). It has to do with the domain specific nature of the problem(s), the user requirements and expectations on the algorithm according to various practical scenarios and probably some other subjective as well as objective factors.

We take two performance metrics to evaluate an algorithm, average error \( \text{avg}_{-}\text{gap} \) and average computational time \( T \).

- \( \text{avg}_{-}\text{gap} \): the difference between the average objective value of the solutions obtained for all the trial runs and the best-known value of the objective function. It serves as a performance metric for measuring the effectiveness of an algorithm.
- \( T \): indicates the average computational time over all the trial runs. It is an indicative measure of the efficiency of an algorithm.

Users are free to use other metrics than \( \text{avg}_{-}\text{gap} \) and \( T \). For example, to evaluate the stability of an algorithm, the user may use the standard deviation of the solutions or the success rate (\( R_s \)), which refers to the ratio of number of trials the algorithm finds the best-known solution to the total number of trials.

Given a candidate algorithm \( \text{algo} \), we design two satisfaction functions \( \xi_1(\text{algo}) \) and \( \xi_2(\text{algo}) \) based on \( \text{avg}_{-}\text{gap} \) and \( T \) to evaluate the algorithm and guide the training process,

\[
\xi_1(\text{algo}) = \frac{1}{\text{avg}_{-}\text{gap} + \epsilon}, \tag{4a}
\]

\[
\xi_2(\text{algo}) = \frac{1}{\text{avg}_{-}\text{gap} \times T + \epsilon}, \tag{4b}
\]

where \( \epsilon \) is a very small real number, which is used to avoid invalid division operation.

It is not difficult to understand that the \( \xi_1(\cdot) \) places all the priority in effectiveness, i.e. the solution quality, and does not consider computational time to be important. On the other hand, in many practical scenarios, users may have concerns on both effectiveness (solution quality) and efficiency of the algorithm. \( \xi_2(\cdot) \) attempts to achieve a balance between effectiveness and efficiency.

C. Trace Update and Initialization

With the algorithm evaluation described above, we can sort the population of candidate algorithms (descending order in satisfaction value) in each iteration of the training. We define \( \text{AlgorithmPop}[k], (k = 0, 1, \ldots, \text{PopSize}) \) as the \( k \)-th best algorithm in the sorted population. Though there are many possible ways to update the trace information based on the generated algorithms, in this work we update the trace information as follows,

\[
\tau_{i+1}^j = \begin{cases} 
\tau_i^j + (\frac{1}{2})^k & \text{if } \text{AlgorithmPop}[k] \text{ covers } \tau_i^j \\
\tau_i^j & \text{else} 
\end{cases}
\]

\[
\xi_1(\text{algo}) = \frac{1}{\text{avg}_{-}\text{gap} + \epsilon}, \tag{4a}
\]

\[
\xi_2(\text{algo}) = \frac{1}{\text{avg}_{-}\text{gap} \times T + \epsilon}, \tag{4b}
\]

where \( \epsilon \) is a very small real number, which is used to avoid invalid division operation.

It is not difficult to understand that the \( \xi_1(\cdot) \) places all the priority in effectiveness, i.e. the solution quality, and does not consider computational time to be important. On the other hand, in many practical scenarios, users may have concerns on both effectiveness (solution quality) and efficiency of the algorithm. \( \xi_2(\cdot) \) attempts to achieve a balance between effectiveness and efficiency.

With the algorithm evaluation described above, we can sort the population of candidate algorithms (descending order in satisfaction value) in each iteration of the training. We define \( \text{AlgorithmPop}[k], (k = 0, 1, \ldots, \text{PopSize}) \) as the \( k \)-th best algorithm in the sorted population. Though there are many possible ways to update the trace information based on the generated algorithms, in this work we update the trace information as follows,

\[
\tau_{i+1}^j = \begin{cases} 
\tau_i^j + (\frac{1}{2})^k & \text{if } \text{AlgorithmPop}[k] \text{ covers } \tau_i^j \\
\tau_i^j & \text{else} 
\end{cases}
\]

(5)
where $r_i^j$ indicates the trace value of the $j$-th branch of a TSN in the $i$-th iteration. ACEM will lay a certain amount of trace value on the root-leaf paths that it covers. From the above equation, it is clear that the better algorithm lays higher trace value on the paths it covers. This way, the competitive features and configurations are emphasized in a cumulative manner and the weak ones are deemphasized accordingly. Therefore in a probabilistic manner, the training/learning process converges to a stable state (e.g. some branches are dominating or the best found algorithm dose not change for a frame of time). This state gives us the optimal or very good configurations of the algorithm for the problem being addressed.

Given the trace update mechanism described above, we simply initialize the trace values by a constant $C$. It is worth noting that the value of $C$ may influence the convergence of the training process. The larger it is, the slower the convergence is, and vice versa. After a series of experiments, the value of $C$ is fine-tuned to be 20.

### D. Simulation Results and Analysis

We test ACEM with both $\xi_1(\cdot)$ and $\xi_2(\cdot)$ based on benchmark problems from QAPLIB [25]. Out of total 25 non-trivial problems, 11 relatively small sized (but non-trivial) instances are chosen as the training set. The remaining 14 medium and large sized problems are used as the testing set. ACEM is implemented using Visual C++ 6.0 and run on an NEC 2.4 GHz PC with 512 MB memory. Each testing is carried out for several trials. It is observed that every training trial obtain either exactly the same or very similar algorithms. For each QAP benchmark instance, 10 trials in training and 20 trials in testing are carried out for average performance. Figure 9 shows the two resulting algorithms through training based on $\xi_1(\cdot)$ and $\xi_2(\cdot)$, respectively.

It is observed that the general structures of the two algorithms (hybrid genetic algorithms) are very similar. However, the differences in the configurations of the two algorithms, which have been highlighted in the figure with bold rectangular blocks, make them perform quite differently. The simulation results are summarized in Table I.

| Table I: Simulation results on QAP benchmark problems |
|------------------|------------------|------------------|
| problem name     | problem size     | $\xi_1(\cdot)$   | $\xi_2(\cdot)$   |
| Training         | avg_gap | $T_{(sec)}$ | $R_e$ | avg_gap | $T_{(sec)}$ | $R_e$ |
| cle25a           | 25      | 0.0085 | 0.0064 | 12.57 | 100%   |
| cle50a           | 50      | 0.0033 | 0.0026 | 8.12  | 100%   |
| nms30            | 30      | 0.0058 | 0.0050 | 11.93 | 100%   |
| yoo50b           | 50      | 0.0065 | 0.0063 | 10.56 | 100%   |
| bms40            | 40      | 0.0051 | 0.0072 | 9.51  | 100%   |
| rws40            | 40      | 0.3281 | 0.3281 | 48.70 | 100%   |
| gms40            | 40      | 0.0065 | 0.0065 | 4.30  | 100%   |
| rws60            | 60      | 0.0024 | 0.0024 | 4.30  | 100%   |
| car100           | 100     | 0.5334 | 0.5334 | 134.90| 100%   |
| car200           | 200     | 0.5334 | 0.5334 | 134.90| 100%   |
| car400           | 400     | 0.5334 | 0.5334 | 134.90| 100%   |
| car800           | 800     | 0.5334 | 0.5334 | 134.90| 100%   |
| car1600          | 1600    | 0.5334 | 0.5334 | 134.90| 100%   |
| car3200          | 3200    | 0.5334 | 0.5334 | 134.90| 100%   |

From Table I, it is shown that through the training based on $\xi_1(\cdot)$, ACEM produces an algorithm that generates very good solutions in both training and testing. It is able to find the best-known solutions for 20 out of 25 instances. It is also observed that, training through $\xi_2(\cdot)$ results in an efficient algorithm as evidenced by the fact that the computational time is significantly less than that of $\xi_1(\cdot)$ while the solution quality shows no significant compromise. Based on $\xi_2(\cdot)$, ACEM produces an algorithm which achieves a good balance between the two performance metrics. This implies that ACEM is able to learn from the user-specialized requirements and in turn produce an algorithm which satisfies the user’s needs.

We further compare the resulting hybrid genetic algorithm using $\xi_1(\cdot)$ with two state-of-the-art algorithms recently reported in the literature for solving the QAP, namely, the compounded genetic algorithm [20] and the population-based iterated local search (ILS) [24]. The comparison is summarized in Table II. It is observed from Table II that the resulting algorithm outperforms the population-based ILS algorithm [24]. This is evidenced by the fact that the hybrid genetic algorithm produces significantly better average solutions for all the instances and higher success rate for most of the instances being considered. Comparison of the hybrid genetic algorithm to the compounded GA [20] for the symmetric QAP problems further highlights the superiority of our algorithm evidenced by competitive solution qualities at significantly lower computational cost (on a slower computer). Considering the fact that our algorithm produces the best (or equal) average solutions (highlighted in bold) in 10 out of total 16 problems being considered, we
We contend that it is ACEM that tunes the algorithm to its full potential in achieving superior performance.

VI. CONCLUSIONS

We proposed and developed Automatic Configuration Engine for Metaheuristics (ACEM) for solving complex combinatorial optimization problems. We implemented ACEM and tested it on QAP benchmark problems. ACEM produced a hybrid genetic algorithm with human-competitive or even better performance. ACEM also succeeded in generating another algorithm which achieves the user-specified requirements on performance. Therefore, we conclude that with ACEM, both the efficiency and effectiveness of the metaheuristic algorithm development for both practical and theoretical purpose are enhanced significantly.

In the future work, we will explore the automatic configuration on more metaheuristics in wider problem domain. We will also investigate deep into the internal evolutionary mechanism in the training process. We believe some sensitivity analysis would help us to better understand the stochastic nature of ACEM. Preliminary results show that the progress of the trace value based on different satisfaction functions is an interesting topic. We will report the research progress in future publications.

REFERENCES


