1-2013

Automated Parameter Tuning Framework for Heterogeneous and Large Instances: Case study in Quadratic Assignment Problem

Linda LINDAWATI
Singapore Management University, lindawati.2008@smu.edu.sg

Zhi Yuan
Singapore Management University, zhiyuan@smu.edu.sg

Hoong Chuin LAU
Singapore Management University, hclau@smu.edu.sg

Feida ZHU
Singapore Management University, fdzhu@smu.edu.sg

Automated Parameter Tuning Framework for Heterogeneous and Large Instances: Case Study in Quadratic Assignment Problem

Lindawati, Zhi Yuan, Hoong Chuin Lau, and Feida Zhu

School of Information Systems, Singapore Management University, Singapore
{lindawati,zhiyuan,hclau,fdzhu}@smu.edu.sg

Abstract. This paper is concerned with automated tuning of parameters of algorithms to handle heterogeneous and large instances. We propose an automated parameter tuning framework with the capability to provide instance-specific parameter configurations. We report preliminary results on the Quadratic Assignment Problem (QAP) and show that our framework provides a significant improvement on solutions qualities with much smaller tuning computational time.

Keywords: automated parameter tuning, instance-specific parameter configuration, parameter search space reduction, large instance parameter tuning

1 Introduction

Good parameter configurations are critically important to ensure algorithms to be efficient and effective. *Automated parameter tuning* (also called *automated algorithm configuration* or *automated parameter optimization*) is concerned with finding good parameter configurations based on training instances. Existing approaches for *automated parameter tuning* fall into two categories: *model-free* and *model-based*. Some *model-free* approaches can handle a large number of numerical and even categorical parameters (for example GGA [1], F-Race [4] and ParamILS [16]). *Model-based* approaches, on the other hand, offers statistical insights into the correlation of parameters with regard to algorithm performance. Examples of the model-based approaches are SPO [2] and SMAC [15].

In this paper, we are concerned with two specific challenges of automated parameter tuning:

1. Heterogeneity. This refers to the phenomenon that different problem instances require different parameter configurations on the same target algorithm to solve. [14] defines "inhomogeneous" instances as those for which algorithm rankings are unstable across instances. They state that inhomogeneous instance sets are problematic to address with both manual and automated methods for offline algorithm configuration. [26] provides two quantitative measures of homogeneity and observes that homogeneity increases

when partitioning instance sets by means of clustering based on observed runtimes. In this paper, we are concerned with the notion of heterogeneity in the sense that instances perform differently when run with different configurations. Table 1 gives an illustration of this phenomenon for benchmark QAP instances on robust tabu search algorithm as presented in [9].

2. Large problem instances. By "large", we mean instances that require a prohibitively long computation time [28]. This notion of largeness is typically tied to the size of the problem instances (i.e. larger problem instances typically take longer time to run), but one needs to be mindful that this quantity varies across problems and target algorithms.

Table 1. Effect of Three Different Parameter Configurations on 4 QAP instances performance. The performance is an average of percentage deviation from optimum or best known solution.

Instances	Configuration 1	Configuration 2	Configuration 3
tai40a	1.4	**1.0**	2.0
tai60a	1.7	**1.6**	2.2
tai40b	9.0	9.0	**0.0**
tai60b	2.1	2.9	**0.3**

There have been approaches that attempt to find instance-specific configurations for a heterogeneous set of instances (see for example, [17, 21, 29]). Unfortunately, finding *instance features* itself is often tedious and domain-specific [8], requiring a discovery of good features for each new problem. Similarly, tuning algorithms for large instances is a frustrating experience, as the tuning algorithm typically requires a large number of evaluations on training instances. This quickly makes automatic tuning suffer computationally prohibitive run time.

In this paper, we attempt to tackle the above challenges by proposing a new automated parameter tuning framework AutoParTune that attempt to bring together components that are helpful for automated parameter tuning. Having a unified software framework allows algorithm designers to readily experiment with different mixes of parameter tuning components in deriving good parameter configurations. Our emphasis in this paper is heterogeneity and large instances, and we briefly describe our approach as follows.

To handle heterogeneous instances, we propose a generic instance clustering technique SufTra that mines generic features from instance search trajectory patterns. For this purpose, we make use of a novel data structure "suffix tree" [7]. A search trajectory is defined as a path of solutions discovered by the target algorithm as it searches through its neighborhood search space [13]. A nice characteristic of our work is that we can obtain these trajectories from the target local search algorithm with minimal additional computation effort. Our approach improves the work of [21] that captures similarity using a single (and relatively short) segment through out the entire sequence, and works only on short and

small number of sequences due to its inherit computational bottleneck. In contrast, our approach is capable of retrieving similarity across multiple segments with linear time complexity. Using a Suffix Tree data structure, our approach can efficiently and effectively form better and tighter clusters and hence improve the overall performance of the underlying target algorithm.

To handle large instances, we propose ScaLa that automatically finds computationally less expensive instances as surrogate to large instances. ScaLa detects similarity among different instances with different runtime using performance-based similarity measures [26]. In this work, we experimentally explore the feasibility of this approach.

We apply our approach on the Quadratic Assignment Problem (QAP), since it is a notoriously hard problem that has been shown to have heterogeneous instances, and whose run time grows rapidly with input size. The major contributions (and thus the flow) of this paper are summarized as follows:

- We propose a new generic automated parameter tuning framework AutoParTune for handling heterogeneous and large instances.
- We present SufTra, a novel technique for clustering heterogeneous instances.
- We present ScaLa that performs runtime analysis for scaling large instances.

2 AutoParTune

AutoParTune is a realization of the concept proposed in [20]. As shown in Fig. 1(a), the framework consists of four components: (I) parameter search space reduction; (II) instance-specific tuning to handle heterogeneous instances; (III) scaling large instances; and (IV) global tuning. The first components are considered as pre-processing steps which can be executed in any combination and in any order. The detail of the AutoParTune subsystems is shown in Fig. 1(b).

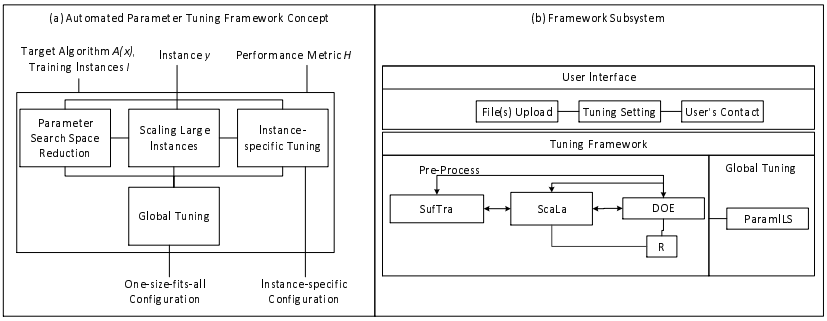

Fig. 1. Automated Parameter Tuning Framework Concept and Infrastructure Overview

Parameter Search Space Reduction. This component allows us to drastically reduce the size of the parameter space before tuning. We implement the

method presented in [6], which is based on Design of Experiment (DOE), a well-established statistical approach that involves experiment designs for the empirical modeling of processes. This work has been presented in [6], and briefly, it consists of three phases: screening, experimentation, and exploitation phase.

Instance-specific Tuning. For the instance-specific tuning, we construct a generic instance-specific parameter tuning: SufTra. The detail will be presented in Section 4.

Scaling Large Instances. For handling large instances, we present a novel technique based on the computational time analysis: ScaLa. The detail will be presented in Section 5.

Global Tuning. There are many global tuning methods in the literature. Here, we embed ParamILS [16] which utilizes Iterated Local Search (ILS) to explore the parameter configuration space in order to find a good parameter configuration for the given training instances. ParamILS has been very successfully applied to a broad range of high-performance algorithms for several hard combinatorial problems with large number of parameters.

3 Target Algorithm and Experiment Setup

To avoid confusion, we refer the algorithm whose performance is being optimized as *target algorithm* and the one that is used to tune it as the *configurator*. As target algorithm, we use the hybrid Simulated Annealing and Tabu Search (SA-TS) algorithm (presented in [22]). It uses the Greedy Randomized Adaptive Search Procedure (GRASP) [30] to obtain an initial solution, and then using a combined Simulated Annealing (SA) [18] and Tabu Search (TS) [5] algorithm to improve the solution. There are four parameters, discrete and continuous, to be tuned as described in Table 2.

In SufTra, we use a set of instances from two generators in [19] for single-objective QAP as in [23]. The first generator generates uniformly random instances where all flows and distances are integers sampled from uniform distributions. The second generator generates flow entries that are non-uniform random values, having the so called real-like structure since it resemble the structure of QAP problems found in practical applications. We generated 500 instances with size from 10 to 150 from each generator and randomly choose 100 as training instances and 400 as testing instances. In ScaLa, we use 120 training instances with size 50, 70, 90, and 150; and 100 testing instances with size 150 from the first generator.

All experiments were performed on a 1.7GHz Pentium-4 machine running Windows XP for SufTra and on a 3.30GHz Core (TM) running Windows 7 for ScaLa . We measured runtime as the CPU time needed by these machine.

4 SufTra: Clustering Heterogeneous Instances

SufTra is premised on the assumption that an algorithm configuration is correlated with its fitness landscape, i.e. configuration that performs well on a problem

Table 2. Parameters for SA-TS on QAP

Parameter	Description	Type	Range	Default
Temp	Initial temperature of SA algorithm	Continuous	[100, 5000]	5000
Alpha	Cooling factor	Continuous	[0.1, 0.9]	0.1
Length	Length of tabu list	Discrete	[1, 10]	10
Limit	maximum number of non-improving iterations prior to intensification strategy (restart the search from best-found solution)	Discrete	[1, 10]	10

instance of certain fitness landscape will also perform well on another instance of a similar topology [24]. Furthermore, since fitness landscape is difficult to compute, it can be approximated by a search trajectory [10, 11] which is deemed a probe through the landscape under a given algorithm configuration. SufTra is an extension of the preliminary work on search trajectory clustering CluPaTra [21] that overcome the following CluPaTra major limitations.

1. **Scalability.**
   Pair-wise sequence alignment is implemented using standard dynamic programming with a complexity $O(m^2)$, where m is the maximum sequence length of the sequences. Hence, the total time complexity for all instances is $O(n^2 \times m^2)$, where n is the number of instances and m is the maximum sequence length. This poses a serious problem for instances with long sequences and when the number of instances is large.
2. **Flexibility.**
   The nature of sequence alignment is to align a segment of sequences pair that gives us the highest alignment score. A matched symbol contributes a positive score (+1), while a gap contributes a negative score (-1). The sum of the scores is taken as the maximal similarity score of the two sequences. However, it is possible that sequences share similarity on more than one segment, especially for long sequences. Sequence alignment is not flexible enough to capture multi-segment alignment with an acceptable time complexity.

SufTra works by transforming search trajectory into a string of symbols based on its solution attributes. A suffix tree is constructed on these strings to extract frequent substrings with length $\geq$ *min_length* and support $\geq$ *min_support*. Substrings may occur in multiple segments along the search trajectory, it allows us to consider **multi-segment** similarities to improve the accuracy of the clusters. Using these frequent substrings as features to cluster the strings, we calculate the similarity and cluster the instances.

4.1 Search Trajectory Representation and Extraction

Suffix Tree Structure. Suffix Tree is a data structure that exposes internal structure of a string for a particularly fast implementation of many important string operations. The construction of a suffix tree proves to have a linear time complexity w.r.t. the input string length [7]. A suffix tree T for an m-character

string S is a rooted directed tree having exactly m leaves numbered 1 to m. Each internal node, other that the root, has at least two children and each edge is labeled with a substring (including the empty substring \$) of S. No two edges out of a node has edge-labels beginning with the same character. To represent suffixes of a set $\{S_1, S_2, ...S_n\}$ of strings, we use a *generalized* suffix tree. *Generalized* suffix tree is built by appending a different end of string marker (which is a symbol not in used in any of the string) to each string in the set, then concatenate all the strings together, and build a suffix tree for the concatenated string [7].

Search Trajectory for Suffix Tree Structure. In building a suffix tree for a search trajectory, we first transform the search trajectory to a sequence of symbols based on its solution's attributes. Each symbol encodes a combination of two solution attributes: (1) position type, based on the topology of the local neighborhood as given in Table 3 [13]; and (2) percentage deviation of quality from optimum or best known. These two attributes are combined; of which the first two digits are the deviation of the solution quality and the last digit is the position type. To handle target algorithms with cycles and (random) restarts, SufTra adds two additional symbols: 'CYCLE' and 'JUMP'; 'CYCLE' is used when the target algorithm returns to a previously-visited position, while 'JUMP' is used when the local search is restarted.

Table 3. Position Types of Solution

Position Type Label	Symbol	<	=	>
SLMIN (strict local min)	S	+	-	-
LMIN (local min)	M	+	+	-
IPLat (interior plateau)	I	-	+	-
SLOPE	P	+	-	+
LEDGE	L	+	+	+
LMAX (local max)	X	-	+	+
SLMAX (strict local max)	A	-	-	+

'+' = present, '-' = absent; referring to the presence of neighbors with larger ('<'), equal ('=') and smaller ('>') objective values

Note that in a search trajectory, several consecutive solutions may have similar solution properties before final improvement and reaching local optimum. We therefore compress the search trajectory sequence to a *Hash String* by removing the consecutive repetition symbols and store the number of repetitions in a *Hash Table* to be used later in pair-wise similarity calculation. Removing consecutive repetition symbols gives us two advantages: (1) it offers greater flexibility over SufTra in capturing more varieties of similarity for symbol patterns between two instances. Two instances may share similar pattern (such as: *14L-5L*) but different number of consecutive symbols, e.g., for $14L$, one has 10 while the other one has 5. And (2) it reduces computational cost for constructing and exploring suffix tree, since the time needed is decided by the sequence length. *Hash String*

is a more compact and shorter representation of the original search trajectory string.

Suffix Tree Construction. We construct the suffix tree using Ukkonen's algorithm [7]. To cover every training instances, we build a single *generalized* suffix tree by concatenating all the *Hash String* together. Length of the concatenate string is proportional to the sum of all the *Hash String* lengths. The Ukkonen's algorithm works by first building an implicit suffix tree containing the first character of the string and then adding successive characters until the tree is complete. Details of Ukkonen's algorithm can be found in [7]. Our Ukkonen's algorithm implementation requires $O(n \times l)$, where n is the number of instances and l is the maximum length of the *Hash String*.

Features Extraction. After constructing the suffix tree, we extract frequent substrings as features. A substring is considered as frequent if it has a length greater than $\min_{length}$ and it occurs in at least $\min_{support}$ number of strings. Min_{length} and $\min_{support}$ value is different for each problem. Fixing the value is not be flexible but running exhaustive search also requires a substantial amount of time. To find optimum (or near-optimum) value, we apply local search, a simple but effective method to provide good enough value in reasonable time. We use local search to move from $\min_{length}$ and $\min_{support}$ initial value to its neighbors by changing either $\min_{length}$ or $\min_{support}$ at each move until it finds optimum cluster. To find $\min_{length}$ and $\min_{support}$ initial value, we run a competition between 5 candidate: (1) Lower bound of $\min_{length}$ and $\min_{support}$; (2) Upper bound of $\min_{length}$ and $\min_{support}$; (3) Middle value between lower and upper bound; (4 and 5) Random values.

4.2 Similarity Score Calculation

After extracting the features, we calculate instance's scores for each feature and construct an instance-feature metric using the following rules:

1. If the instance does not contain the feature, the score is 0.
2. Else the score is calculated by summing number of repetition for each symbol in feature from previously constructed *Hash Table*. A frequent substring may occur few times in one string. We calculate the score for each occurrence and choose the maximum score as a score for instance-feature metric.

Using the metric, we calculate similarity for each pair of instances by applying cosine similarity. Cosine similarity has been widely used to measure similarity between two vectors by measuring cosine angle between them [7]. Cosine similarity result is equal to 1 when the angle is 0, and it is less than 1 when the angle is of any other value. Cosine similarity is formulated as follows.

$$similarity = \frac{\sum_{i=0}^{n}(Inst_1(feature_i) \times Inst_2(feature_i))}{\sqrt{\sum_{i=0}^{n} Inst_1(feature_i)^2} + \sqrt{\sum_{i=0}^{n} Inst_2(feature_i)^2}} \quad (1)$$

with $Inst_1(feature_i)$ and $Inst_2(feature_i)$ as score from instance-feature metric for Instance 1 and 2.

4.3 Clustering

Similar to [21], we cluster the instances bys a well-known clustering approach AGNES [12] with L method [25]. AGNES or AGglomerative NESting is a hierarchical clustering approach that works by creating clusters for each individual instance and then merging two closest clusters (i.e., a pair of clusters with the smallest distance) resulting in fewer clusters of larger sizes until all instances belong to one cluster or a termination condition is satisfied (e.g. a prescribed number of clusters is reached). We implement the L method [25] to automatically find the optimal number of clusters, which works by using the evaluation graph where the x-axis is the number of clusters and the y-axis is the value of the evaluation function at x clusters. In this paper, we use the average distance among all instances in two different clusters as the evaluation function. L method fits the curve in the evaluation graph into two lines and chooses the intersection point between these two lines as the optimal number of clusters.

4.4 Experimental Result

To evaluate SufTra's effectiveness, we first compared the time needed (in seconds) for SufTra and CluPaTra to form the clusters in training phase and to map the testing instances in testing phase. Table. 4 (I) shows the result. From the table, we observe that SufTra is 18 times faster then CluPaTra.

Next, we compared the *target algorithm* performance using parameter configuration from SufTra, CluPaTra and ISAC as well as the one-size-fits-all configurator ParamILS. Since ISAC requires problem-specific features, we used 2 features: *flow dominance* and *sparsity* of flow metric which is believed to have significant influence on the performance [27].

For the three instance-specific methods, we used the same one-size-fits-all configurator, ParamILS [16]. Since ParamILS works only with discrete parameters, we first discretized the values of the parameters. We measured the performance as the average of percentage deviation from optimum or best known solution. We set the cutoff runtime of ParamILS to 100 second. For CluPaTra and SufTra, we allowed each configurator to execute the target algorithm for a maximum of two CPU hours for each cluster. To ensure fair comparison, we set the time budget for ISAC and ParamILS to be equal to the total time needed to run SufTra. For unbiased evaluation, we used a 5-fold cross-validation [12] and measured the average performance over all folds. We also performed a statistical test (*t-test*) on the significance of our result where a *p-value* below 0.05 is deemed to be statistically significant.

In Table. 4 (II), we show the performance comparison results. From the table, we observe that SufTra performs better on training and testing instances compare to other approaches. But the result for training instances is not statistically significant compare to ISAC.

Table 4. QAP Experiment Result

	Training	Testing
I. Computational Time		
CluPaTra	1,051 s	2,718 s
SufTra	**56 s**	**146 s**
II. Performance Result		
ParamILS	1.07	2.12
CluPaTra	0.87	1.54
ISAC	0.83	1.21
SufTra	**0.81**	**1.16**
p-value*	0.061	0.042

*based on statistical test on ISAC and SufTra

5 ScaLa: Scaling Large Instances

As mentioned in the introduction, tuning on instances that requires long computation time to solve is a challenging task. These instances are referred to as *hard* or *large* instances throughout this section. To make automatic tuning applicable for such hard instances, one idea is to tune on easier instances [3, 28]. Styles et al. proposed in [28] to run multiple tuning processes on small instances, validate the independently tuned configurations on medium instances, and use the best validated configuration for solving the large instances. Our current work takes a different direction. The goal is to tune on easy instances with short runtime, such that the tuned configuration performs similarly on hard instances with long runtime. In order to realize this goal, a number of questions have to be addressed:

1. how to measure similarity among different instances?
2. Does there exist similarity between easy instances and hard instances at all?
3. Do good configurations on easy instances perform well on similar hard instances?

Each of the three questions above will be answered in one of the following subsections.

5.1 Measuring instance similarity

To answer question 1, there exist two different approaches to finding similarities among instances. One is based on problem-specific instance features, e.g. instance size, fitness distance correlation, search trajectory patterns [21] etc. Both instance-specific configurators ISAC [17] and CluPaTra [21] cluster instances based on problem-specific features, then tune on each cluster, and confirm that tuning on separate clustered instance set leads to better performance than tuning on all instances. Unlike our approach in this work, they didn't take into account the computation time. Another approach is based on empirical performance [26].

Scheider et al. introduced in [26] two measures, a *ratio measure* and a *variance measure*, for measuring instance similarity based on relative performance of different algorithms (or same algorithm with different configurations). However, the performance-based similarity measure depends on two folds: the computation time and the solution quality. Although [26] considered computation time, but didn't consider scaling among different instances by, e.g. considering different solution quality threshold. In this work, we adopt the performance-based similarity measures proposed in [26], more specifically, the *variance measure* that is described in more details in the next section, and use them to find similarities among different instances with different runtime.

5.2 Finding similarities between hard and easy instances

To answer question 2, we set up experiments to test the hypothesis whether hard instances could be similar to easy instances at all given different computation time. Unlike in SufTra (Sec. 4) where we try to separate heterogeneous instances, here the goal is to join instances with different features, given different runtime. We take QAP as our target problem, and an implementation of SA-TS algorithm as our target algorithm (see Sec. 3 for a description and parameter ranges). Our preliminary experiments consider only instance size as a measure of instance "hardness". 30 instances are generated for each of the four instance sizes 50, 70, 90, and 150. The SA-TS has four parameters as described in Table 2. In order to use the performance-based measure, 100 parameter configurations are sampled uniformly within the parameter range. Each parameter configuration runs once on each instance. The solution cost $c_\theta(n, t_n)$ of a configuration $\theta \in \Theta$ on an instance size $n \in N = \{50, 70, 90, 150\}$ with a given runtime t_n is computed by taking the mean solution cost across the 30 instances with size n, and $C_\Theta(n, t_n) = \{c_\theta(n, t_n), \theta \in \Theta\}$. For each instance size n, a set of runtime T_n is determined as follows: let minimum runtime $t_{min} = 0$, maximum runtime t_{max} takes value of the maximum natural stopping time of the algorithm (no restart), and T_n takes values in a logarithmically spaced sequence between t_{min} and t_{max}, excluding t_{min}. Following [26], we perform a standardized z-score normalization for each cost vector $C_\Theta(n, t_n)$, and use the variance measure

$$Q_{var}(\Theta, N', T_{N'}) = \frac{1}{|\Theta|} \sum_{\theta \in \Theta} Var(c_\theta(N', T_{N'})), \text{ for } N' \subseteq N \tag{2}$$

for measuring similarity (more precisely, dissimilarity) among different pairs of $(n, t_n) \in (N, T_N)$. Based on this measure Q_{var}, instances of different size and computation time can be clustered with the goal of optimizing similarity of the resulting subsets. A classical clustering approach Hierarchical Agglomerative Clustering or AGNES [12] is adopted in our preliminary experiments (an alternative clustering method K-mean also gives very similar clustering result). For illustrative purpose, 5 logarithmic time intervals for each instance size n are used, excluding t_{min}, this makes $|T_n| = 4$. The clustering results are shown in Fig. 2. Interestingly, the most similar two subgroups turned out to be the longest

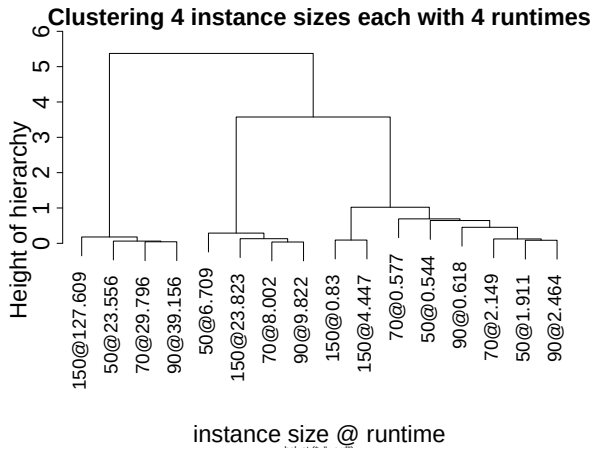

Fig. 2. Clustering four different instance sizes each with four different computation times by hierarchical agglomerative based on variance measure.

runtime (natural stopping time) of each of the four instance sizes $n \in N$, and the second longest logarithmic runtime level of each $n \in N$. More specifically, the four (n, t_n)-pairs $(50, 23.6), (70, 29.8), (90, 39.2), (150, 127.6)$ form the most similar group, while $(50, 6.7), (70, 8.0), (90, 9.8), (150, 23.8)$ comprise the second most similar group. In the two shorter levels of runtime, the similarities across the four instance sizes are less obvious. Nevertheless, this interesting clustering result confirms our hypothesis raised in question 2: using performance-based similarity measure, given the right runtime, different instances, easy or hard, can become similar to each other.

5.3 Solving hard instances by tuning on easy instances

How can automatic tuning benefit from this automatically detected instance similarity? One straightforward follow-up idea is to use the best parameter configuration tuned on easy instances with short runtime to solve similar hard instances with long runtime. However, it remains unjustified that how good these tuned-on-easy parameter configurations are, compared with, for example, parameter tuned directly on instances of the same size with the same runtime. In this experiment, two most similar groups of size-runtime pairs (see Fig. 2 of Sec. 5.2) are used: the first group includes (50, 6.709), (70, 8.002), (90, 9.822), and (150, 24.823); the second group includes (50, 23.556), (70, 29.796), (90, 39.156), (150, 127.609). For each of the two groups, two sets of experiments are set up: 1) tuned by `oracle`: as a quick proof-of-concept, we take the best configuration from 100 configurations based on 30 instances on each instance size as found in Sec. 5.2, and test them on another 100 testing instances of size 150 with corresponding runtime; 2) tuned by ParamILS [16]: we tune the target algorithm using three independent runs of ParamILS for each instance size using the size-runtime pairs

mentioned above, each run was assigned maximum 300 calls of target algorithm on new randomly generated training instances, and each tuned configuration is tested on 100 same testing instances as in 1). The second experiment set is to test generalizability of the similarity information detected in Sec. 5.2. The goal is to see how good these best configurations tuned on small instances such as 50, 70, and 90 with shorter runtime, compared with the best configuration tuned on instance size 150, when tested on instance size 150 with the same runtime.

The results are listed in Table 5. The results confirm that, firstly as expected, a large amount of tuning time is saved by tuning on small instances, ranging from 59 to 81% in our experiments; and secondly, in general, parameter configurations tuned on smaller instances with shorter runtime don't differ significantly from the ones directly tuned on large instances, as long as similarities between them can be found. In both groups in both experiment sets, there is no statistical difference between the configurations tuned on 50, 70, 90, and 150, tested by Wilcoxon's rank-sum test. In the first experiment set tuned by `oracle`, configurations tuned on size 70 and 90 sometimes perform even better than tuned on 150. The mean performance difference from the tuned-on-150 configuration in the first group is usually less than 0.1%, and even less than 0.01% in the second group. In the second experiment set tuned by ParamILS, although configuration tuned on size 150 performs best, the difference is not significant: the mean performance difference is usually less than 0.1% in the first group, and less than 0.05% in the second group. This shows the similarity information detected from Sec. 5.2 can be actually generalized to tuners with different training instances. As reference, the performance of the default parameter configuration (listed in Table 3) is presented in Table 5, and it is statistically significantly outperformed by almost all the above tuned configurations in both groups, which proves the necessity and success of tuning process. We also include as reference the best configuration tuned on instance size 150 with runtime 23.556 (127.609) seconds to be tested on instance size 150 with different runtime, i.e. 127.609 (23.556) seconds, respectively (in row 150' of Table 5). Although the tuning and testing instances are of the same size, different runtime makes a great performance difference, resulting in almost one order of magnitude worse than tuning on the small instances with appropriate runtime. The 150' performance is statistically significantly worse than all the above tuned configurations belonging to the same group, and it is even significantly worse than the default configuration in the second group. This contrasts with the fact that the difference among the similar size-runtime pairs (the first four rows of Table 5) is indeed very minor, and it also shows the risk of tuning on algorithm solution quality without assigning the right runtime, which in fact proves the necessity of our automatic similarity detection procedure in ScaLa.

6 Conclusion and Future Work

In this paper, we proposed an automated parameter tuning framework for heterogeneous and large instances and tested it on Quadratic Assignment Problem

Table 5. Results for the performance of the best parameter configurations tuned on sizes 50, 70, 90, 150, and tested on instances of size 150. Two most similar groups of size-runtime pairs (see text or Fig. 2) are used. Two experiment sets are presented, `oracle` and ParamILS (see text). Each column of `%oracle` and `%ParamILS` shows the mean percentage deviation from the reference cost. In each column, $+x$ ($-x$) means that the tuned configuration performance is x% more (less) than the reference cost. `%time.saved` shows the percentage of tuning time saved comparing with tuning on instances of size 150. The performance of default parameter configuration is shown in row "def.". The last row 150' used the best parameter configuration tuned on instance size 150 with runtime 127.609 (23.556) seconds, and tested on instance size 150 with runtime 23.556 (127.609) seconds, respectively. Results marked with † refers to statistically significantly worse results compared to tuned-on-150 using Wilcoxon's rank-sum test.

	23.556 seconds			127.609 seconds		
tuned.on	%oracle	%ParamILS	%time.saved	%oracle	%ParamILS	%time.saved
50	−0.48	−0.047	72	−0.048	−0.048	81
70	−0.65	−0.053	66	−0.060	−0.027	76
90	−0.61	−0.093	59	−0.057	−0.040	69
150	−0.58	−0.151	0	−0.060	−0.070	0
def.	$+1.17^{\dagger}$	$+0.150^{\dagger}$	-	$-0.008^{\dagger}$	−0.024	-
150'	$+1.16^{\dagger}$	$+0.195^{\dagger}$	-	$+0.232^{\dagger}$	$+0.208^{\dagger}$	-

(QAP). We construct SufTra for tuning heterogeneous instances and ScaLa for large instances. We verify SufTra's performance and observed a significant improvement compared to a vanilla one-size-fits-all approach (ParamILS) and other generic instance-specific approach CluPaTra. We claim that: (1) SufTra is a suitable approach for instance-specific configuration that significantly improves the performance with minor additional computational time; and (2) SufTra has overcome CluPaTra limitations with a new efficient method for feature extraction and similarity computation using suffix tree. In the development of ScaLa, we use the performance-based similarity measure and clustering technique to automatically detect and group similar instances with different sizes by assigning different runtime, such that one can tune on easy instances with much less runtime and apply the tuned configuration to solve hard instances with long runtime. This greatly reduces computation time required when tuning large and computationally-hard instances. Through our preliminary experiments, we empirically show that easy instances and hard instances can be similar when given the right runtime, and in such case, the good configurations tuned on easy instances can also perform well on hard instances.

Up to this stage of our work, the SufTra and ScaLa are not yet integrated. In near future, we plan to integrate those two components on the AutoParTune framework, in particular, we plan to integrate more problem features into ScaLa apart from instance size. As future works on SufTra, we will investigate how to generate clusters from population-based-algorithm using generic features pertaining to population dynamics, since currently SufTra can only be applied to

target algorithms which are local-search-based due to the search trajectory. On the other hand, ScaLa is still an actively ongoing work. Future works include largely extending the amount of experiments, consider also testing on problems other than QAP, and extend our studies to other state-of-the-art algorithms. The correlation between computation time and instance size may be algorithm-specific, therefore, an automatic approach to detecting it is practically valuable. Our current approach is still a proof-of-concept, since it is computationally expensive for computing the performance-based measure. In future work, we plan to investigate how to reduce the computation expenses by, e.g. taking fewer instances and fewer but good configurations found during the tuning process. In particular, we plan to investigate the possibility of predicting the "right" runtime for an unseen instance such that it is similar to a known group of instances.

Acknowledgments. We sincerely thank Saifullah bin Hussin, Thomas Stützle, Mauro Birattari, Matteo Gagliolo for valuable discussion on scaling large instances.

References

[1] C. Ansótegui, M. Sellmann, and K. Tierney. A Gender-based Genetic Algorithm for the automatic configuration of algorithms. In *15th international Conference on Principles and Practice of Constraint Programming*, pages 142–157, 2009.

[2] T. Bartz-Beielstein, C. Lasarczyk, and M. Preuss. Sequential parameter optimization. In *Congress on Evolutionary Computation 2005*, pages 773–780. IEEE Press, 2005.

[3] M. Birattari, M. Gagliolo, Saifullah bin Hussin, T. Stützle, and Z. Yuan. Discussion in IRIDIA coffee room, October, 2008.

[4] M. Birattari, Z. Yuan, P. Balaprakash, and T. Stützle. F-Race and Iterated F-Race: An overview. *Experimental methods for the analysis of optimization algorithms*, pages 311–336, 2010.

[5] F. Glover. Tabu Search Part I. *ORSA Journal on Computing*, 1:190–206, 1989.

[6] A. Gunawan, Hoong Chuin Lau, and Lindawati. Fine-tuning algorithm parameters using the design of experiments approach. In *LNCS: 5nd Learning and Intelligent OptimizatioN Conference*, 2011.

[7] D. Gusfield. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, 1997.

[8] I. Guyon, S. Gunn, M. Nikravesh, and L.A. Zadeh, editors. *Feature Extraction: Foundations and Applications*. Springer, 2006.

[9] S. Halim and Y. Yap. Designing and Tuning SLS through animation and graphics an extended walk-through. In *Stochastic Local Search Workshop*, 2007.

[10] S. Halim, Y. Yap, and H.C. Lau. Viz: A visual analysis suite for explaining local search behavior. In *19th ACM symposium on User Interface Software and Technology*, pages 57–66, 2006.

[11] S. Halim, Y. Yap, and H.C. Lau. An integrated white+black box approach for designing and tuning Stochastic Local Search. In *LNCS: 13th International Conference on Principles and Practice of Constraint Programming*, pages 332–347, 2007.

[12] J. Han and M. Kamber. *Data Mining: Concept and Techniques, 2nd Edition*. Morgan Kaufman, San Francisco, 2006.

[13] H.H. Hoos and T. Stützle. *Stochastic Local Search: Foundation and Application*. Morgan Kaufman, San Francisco, 2004.
[14] F. Hutter, H. Hoos, and K. Leyton-Brown. Tradeoffs in the empirical evaluation of competing algorithm designs. *Annals of Mathematics and Artificial Intelligenc (AMAI), Special Issue on Learning and Intelligent Optimization*, 60:65–89, 2011.
[15] F. Hutter, H.H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *LNCS: 5nd Learning and Intelligent OptimizatioN Conference*, 2011.
[16] F. Hutter, H.H. Hoos, K. Leyton-Brown, and T. Stützle. ParamILS: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306, 2009.
[17] S. Kadioglu, Y. Malitsky, M. Sellmann, and K. Tierney. ISAC: Instance-specific algorithm configuration. In *19th European Conference on Artificial Intelligence*, 2010.
[18] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by Simulated Annealing. *Science*, 200:671–680, 1983.
[19] J. Knowles and D. Corne. Instance generators and test suites for the multiobjective Quadratic Assignment Problem. In *LNCS: Evolutionary Multi-Criterion Optimization 2003*, 2003.
[20] H.C. Lau and F. Xiao. Enhancing the speed and accuracy of automated parameter tuning in heuristic design. In *8th Metaheuristics International Conference*, 2009.
[21] Lindawati, H.C. Lau, and D. Lo. Instance-based parameter tuning via search trajectory similarity clustering. In *LNCS: 5nd Learning and Intelligent OptimizatioN Conference*, 2011.
[22] K.M. Ng, A. Gunawan, and K.L. Poh. A hybrid algorithm for the quadratic assignment problem. In *International Conf. on Scientific Computing*, pages 14–17, 2008.
[23] G. Ochoa, S. Verel, F. Daolio, and M. Tomassini. Clustering of local optima in combinatorial fitness landscape. In *LNCS: 5nd Learning and Intelligent OptimizatioN Conference*, 2011.
[24] C.R. Reeves. Landscapes, operators and heuristic search. *Annals of Operations Research*, 86(1):473–490, 1999.
[25] S. Salvador and P. Chan. Determining the number of clusters/segments in hierarchical clustering/segmentation algorithms. In *16th IEEE International Conference on Tools with Artificial Intelligence*, pages 576–584, 2004.
[26] M. Schneider and H.H. Hoos. Quantifying homogeneity of instance sets for algorithm configuration. In *LNCS: 6nd Learning and Intelligent OptimizatioN Conference*, 2012.
[27] T. Stützle and S. Fernandes. New benchmark instances for the QAP and the experimental analysis of algorithms. In *LNCS: Evolutionary Computation In Combinatorial Optimization*, 2004.
[28] J. Styles, H.H. Hoos, and M. Muller. Automatically configuring algorithms for scaling performance. In *LNCS: 6nd Learning and Intelligent OptimizatioN Conference*, 2012.
[29] L. Xu, H.H. Hoos, and K. Leyton-Brown. Hydra: Automatically configuring algorithms for portfolio-based selection. In *Conference of the Association for the Advancement of Artificial Intelligence (AAAI-10)*, 2010.
[30] L. Yong, P.M. Pardalos, and M.G.C. Resende. A greedy randomized adaptive search procedure for the quadratic assignment problem. *Quadratic Assignment and Related Problems, DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 16:237–261, 1994.