Instance-specific Parameter Tuning via Constraint-based Clustering

Lindawati and **Hoong Chuin LAU** and **Feida ZHU** [1]

Abstract. The performance of heuristic algorithm, is highly dependent on its parameter configuration. To automatically obtain good parameter configurations, we propose SufTra, a novel approach for instance-specific parameter tuning that utilizes the *Suf*fix tree data structure to represent *Tra*jectories of a Local Search algorithm. SufTra extracts compact features from search trajectories using Suffix Tree and filters these features using user specified-constraints. Then, it clusters the problem instances and computes the parameter configurations. Given an arbitrary testing instance, we obtain a suitable parameter configuration by mapping it to these clusters. Experimental evaluations of our approach on the Quadratic Assignment Problem (QAP) show that our approach offers significant improvement over existing parameter tuning algorithms. We also analyse the cluster quality, demonstrating an almost perfect match between our cluster results with the existing natural classifications.

1 Introduction

Good parameter configurations are critically important to ensure heuristic algorithms to be efficient and effective. Existing approaches for *automated parameter tuning* (also called *automated algorithm configuration* or *automated parameter optimization*) 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 [2], F-Race [4] and ParamILS [9]). *Model-based* approaches, on the other hand, offers statistical insights into the correlation of parameters with regard to algorithm performance. A recent example of the model-based approach is SMAC [8].

The challenge with parameter tuning is that different problem instances require different parameter configurations [10, 12, 19]. One way to enhance model-based approaches is to incorporate *instance features* to produce instance-specific parameter configurations. Unfortunately, finding *instance features* itself is often tedious and domain-specific, requiring a re-examination of features for each new problem. The research problem is to discover a **generic instance-specific automated parameter tuning** scheme that can perform as well as those exploiting problem-specific features.

In this work, we propose an approach to address this problem based on data mining and machine learning concepts. We focus our attention on tuning local search algorithms. In essence, we perform clustering of training instance's search trajectories, which is defined as a path of solutions discovered by the target algorithm as it searches through its neighborhood search space [7], and perform tuning on the clusters. For this purpose, we make use of a powerful data structure, namely suffix tree [5]. The nice characteristic of our work is that we can obtain these trajectories from the target local search algorithm with minimal additional computation effort.

Motivated by previous work [11] where human constraints give significant improvement to its solution, we also involve user-specified constraints to guide the clustering process. Although in some well-known problems such as QAP, user can specify the constraints very easily, for other problems, user-specified constraints are often hard to enumerate. Hence, we design our approach to anticipate partial (rather small) or no user-specified constraints. Note that user-specified constraints are different from instance features.

It is interesting to note that our approach does not make use of an explicit formulation (such as linear or Gaussian regression) that maps instances to clusters, which may be very hard if not impossible to derive. Instead, we exploit the *rich instance-specific* search trajectories as a proxy for the fitness landscape which is correlated with algorithm performance [15]. Instances are clustered based on these generic features using predictive modeling. This form of clustering preserves *rich features* that represent the individual instances within it.

Our approach improves the work of [12] 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 inherent computational bottleneck. In contrast, our approach is capable of retrieving similarity across multiple segments with linear-time complexity. Using a Suffix Tree data structure and user-specified constraints, our approach can efficiently and effectively form better and tighter clusters and hence improve the overall performance of the underlying target algorithm.

We conduct experiments on the Quadratic Assignment Problem (QAP). The experiments show that our approach offers encouraging results for both cluster quality and overall performance compared against existing approaches. For the overall runtime, our approach is significantly (more than ten times) faster compared to the approach in [12] when the search trajectories are long and the number of training instances is large.

2 Preliminaries

To avoid confusion, we refer the algorithm whose performance is being tuned/configured as the *target algorithm* and the one that is used to tune/configure it as the *configurator*. We measure the target algorithm performance based on the quality of their solutions. We define function $\mathcal{H}$ as follows.

Definition 1 (Performance Metric [$\mathcal{H}$]) *Let i be a problem instance, and $\mathcal{A}_{\mathbf{x}}(i)$ be the objective value of the corresponding solution for instance i obtained by a target algorithm $\mathcal{A}$ when executed*

[1] Singapore Management University, Singapore, email: lindawati.2008, hclau, fdzhu@smu.edu.sg

under configuration **x**. *Let $OPT(i)$ denote the best known value for instance* i. *$\mathcal{H}_x(i)$ is formulated as: $\mathcal{H}_x(i) = \frac{|OPT(i) - A_{\mathbf{x}}(i)|}{OPT(i)}$*

For benchmark instances with known global optimum value, we use the known global optimum value as its $OPT(i)$, while for new instances, we use the target algorithm's best solution. Using performance metric $\mathcal{H}$, we define the instance-specific parameter tuning problem as follows.

Definition 2 (Instance-Specific Parameter Tuning [ISPT]) *Given a set of instances I, a parameter configuration space Θ for a target algorithm $\mathcal{A}$ and a performance metric $\mathcal{H}$, the* ISPT *problem is to find a parameter configuration* **x** $\in \Theta$ *for each $i \in I$ such that $\mathcal{H}_x(i)$ is minimized over Θ.*

Instead of finding a parameter configuration for each problem instance, the ISPT problem can be approximated in a cluster-based manner in which problem instances are grouped into clusters and a parameter configuration is computed for each cluster [10, 12]. In this paper, we aim to solve the ISPT problem with user-specified constraints on the resulting clusters. In real-world applications, these user-specified constraints often represent domain knowledge to guide the clustering process. In particular, we consider a simple and natural set of constraints, each of which specifies whether a pair of data samples must belong to the same cluster, which are called *must-link* constraints (M_{link}), or must belong to different clusters, which are called *cannot-link* constraints (C_{link}) [3, 20].

Hence, we define the problem of constrained cluster-based instance-specific parameter tuning as follows.

Definition 3 (Constrained Cluster-based Instance-Specific Parameter Tuning [CC-ISPT]) *Given a set of instances I, a parameter configuration space Θ for a target algorithm $\mathcal{A}$, a performance metric $\mathcal{H}$, a set of user-specified constraints C, the* CC-ISPT *problem is to find a clustering π of all instances of I and a parameter configuration* **x** $\in \Theta$ *for each cluster of π such that (I) π satisfies all the constraints in C; and (II) the average $\mathcal{H}_x(i)$ for each cluster is minimized over Θ.*

If there is no user-specified constraints, user-specified constraints is considered as an empty set. We assume that there is no contradiction between M_{link} and C_{link}.

3 Related Work

Various approaches for *one-size-fits-all configurator* have been proposed in the literature which divided into two categories: *model-free* and *model-based*. For model-free approaches, F-Race and its variants [4] work by using statistical model selection to evaluate a set of candidate configurations and discarding statistically bad configurations. Other model-free approaches that can handle large number of parameters are ParamILS [9] and GGA [2]. ParamILS applies an iterated local search that uses an adaptive capping technique to speed up the search process. GGA (Gender-based Genetic Algorithm) uses a genetic algorithm which divides the parameter candidate set into two groups and applies a different selection method for each group. For model-based approaches, CALIBRA [1] combines statistical experimental design with local search, and this approach handles upto five parameters. A recent work SMAC [8] constructs predictive performance models to focus attention on promising regions of a design space. One common shortcoming of the above approaches is that they provide a **one-size-fits-all** configuration for all instances, which may not perform well on large and diverse instances.

Three recent approaches for instance-specific tuning is Hydra [19], ISAC [10] and CluPaTra [12]. Hydra works by combining automated parameter tuning and portfolio-based algorithm selection. It automatically builds a set of solvers with complementary strengths by iteratively configuring new algorithms to be used in its portfolio. ISAC and CluPaTra work by dividing instances into clusters based on feature(s) similarity and tuning the parameter configuration for each cluster. The main difference between ISAC and CluPaTra is that ISAC uses **problem-specific features** while CluPaTra uses a **generic feature**. For a given problem, ISAC uses different problem-specific features which require in-depth understanding of the problem. For example, in [10], 8 features are used for Set Covering Problem (SCP) and 9 features for Mixed Integer Programming Problem (MIP). CluPaTra uses a generic feature derived from search trajectories to perform clustering, and then tune the parameters for each cluster.

4 Solution Approach

Our approach follows the framework of CluPaTra [12] that makes use of the search trajectories. As CluPaTra uses sequence alignment to calculate similarity, it suffers from the following two limitations.

1. **Scalability.**
   In CluPaTra, 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 search trajectories and when the number of instances is large.
2. **Flexibility.**
   The nature of sequence alignment is to align a pair of sequence segments 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 segments, especially for long sequences. Sequence alignment is not flexible enough to capture multiple-segment alignment with an acceptable time complexity.

To overcome these limitations and achieve better overall performance, we propose a new algorithm called SufTra that uses a compact and rich data structure, namely Suffix Tree, and design a more efficient method to calculate similarity.

SufTra addresses CluPaTra's limitations as follows: (1) Scalability: We propose a linear time algorithm for both Suffix Tree construction and traversal; and (2) Flexibility: We generate compact patterns from search trajectories and use it as features. The patterns may occur in multiple segments along the search trajectory, so suffix trees enable us to consider multiple-segment similarities to improve the accuracy of the clusters.

We further improve the cluster quality by employing user supervision and proposing a new classification method to map testing instances to clusters. This method enables us to generate more accurate mapping in a shorter computation time. In the overall, SufTra runs in linear-time, which is an order of magnitude improvement and translates to a significantly faster method, compared to CluPaTra (that runs in quadratic time). Furthermore, we show experimentally that the quality of clusters outperform those of CluPaTra

4.1 SufTra Framework Overview

The SufTra framework works in two phases: training and testing. The training phase works as follows:

1. Feature Extraction. We extract a set of features F from search trajectories using a suffix tree and remove insignificant features with respect to M_{link} and C_{link}.
2. Similarity Calculation. We calculate similarity scores using the extracted features.
3. Clustering. We cluster the instances using AGNES (AGglomerative NESting).
4. Parameter Configuration. We run an existing one-size-fits-all configurator to obtain the best configuration for each cluster created.

In the testing phase, we use the knowledge from the training phase to return instance-specific configuration(s) for testing instances. This phase is usually performed online. To achieve this, we design a new method for fast and accurate testing instance mapping. Our proposed method consists of two steps:

1. Signature Construction. We construct the signatures for each cluster. This step is run once, and can be performed offline.
2. Cluster Mapping. For an arbitrary testing instance, we match its search trajectory to the cluster's signature and return parameter configuration from the best-matching cluster's as its parameter configuration. This step will be performed online.

Thus, our framework makes use four major components: feature extraction, similarity calculation, clustering and signature extraction. The details of these components are given as follows.

4.2 Feature Extraction

To generate its features, SufTra mines patterns from search trajectories. To mine compact and important patterns, we select pattern that has significant length and appear in a sufficient number of instances [6]. We also filter the patterns using user-specified constraints.Hence, we define the features as follows.

Definition 4 (Feature [$\mathcal{F}$]) *Let I be a set of problem instance, S be a set of search trajectories for all instance in I, min_{length} be a minimum length threshold, $min_{support}$ be a minimum support threshold, C be a set of user-specified constraints. $\mathcal{F}$ is defined as a set of distinct segments from S that: (I) has a length greater than min_{length}; (II) occurs in at least $min_{support}$ number of instances; and (III) has high score for discriminating between all pairs of instances with respect to C.*

The steps on Feature Extraction are as follows.

4.2.1 Search Trajectory Representation

Search trajectory is defined as a path of solutions discovered by the target algorithm $\mathcal{A}$ as it searches through the neighborhood search space [7]. We represent a search trajectory as a directed sequence of symbols before constructing suffix tree. The steps for converting a search trajectory into a string are as follows:

- Record the instance's search trajectory by running the target algorithm using a random parameter configuration.
- Convert the search trajectory to a sequence based on its solution's attributes. Each solution in search trajectory is represented as a symbol which encodes a combination of two solution attributes: (1) percentage deviation of quality from OPT (as defined in Definition 1); and (2) position type, based on the topology of the local neighborhood as given in Table 1 [7]. 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, it adds two additional symbols: 'C' and 'J'; 'C' is used when the target algorithm returns to a previously-visited position (cycle), while 'J' is used when the local search is restarted (jump). An example of a search trajectory sequence is *14L-14L-14L-14L-14L-14L-14L-14L-14L-14L-14L-14L-14L-04M-J-24L-14L-14L-14L-14L-14L-14L-14L*, where *14L* represent the solution that has percentage deviation 14% and position type LEDGE.

Table 1. 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 neighbor solutions with larger ('$<$'), equal ('=') and smaller ('$>$') objective values

- Construct a *Hash Table* to store number of repetitions. Note that in a search trajectory, several consecutive solutions may have similar solution properties before the final improvement and reaching local optimum. We therefore compress the search trajectory sequence to a *Hash String* by removing the consecutive repetition symbols and storing the number of repetitions in a *Hash Table* (to be used later in the similarity score calculation). Removing consecutive repetition symbols gives us two advantages: (1) it offers greater flexibility in capturing more varieties of similarity for symbol patterns between two instances. Two instances may share similar patterns but a different number of consecutive symbols, e.g., for a particular symbol $14L$, in one instance it may occur repeatedly for 10 times, while in the other instance it may occur repeatedly for 5 times. And (2) it reduces computational cost for constructing and traversing the suffix tree, since the time needed is decided by its length. *Hash String* is a more compact and shorter representation of the original search trajectory sequence. An example of *Hash String* is *14L-04M-J-24L-14L*.
- Convert the symbol for each solution to one single character and concatenate it into a string (*Hash String*).

4.2.2 Suffix Tree Construction

The suffix tree is a data structure that exposes the internal structure of a string for a particularly fast implementation of many important string operations. Suffix trees are used to solve exact and inexact matching problems in linear time and are widely used in substring problems [5]. The construction of a suffix tree proves to have a linear time complexity w.r.t. the input string length [5].

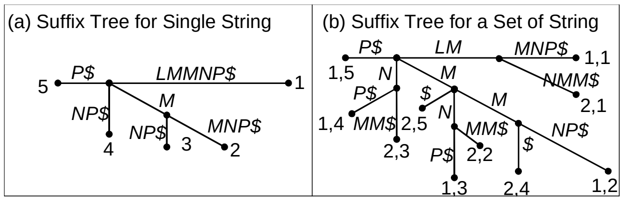

Figure 1. (a) Suffix Tree Example for string S_1 ($LMMNP$), number at a leaf indicates the starting position of the suffix in that string; (b) Generalized Suffix Tree Example for strings S_1=$LMMNP$ and S_2=$LMNMM$, the first number at a leaf indicates the string and the second number indicates the starting position of the suffix in that string.

A suffix tree T for a string S with m length 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 strings) to each string in the set, then concatenate all the strings together, and build a suffix tree for the concatenated string [5]. An example of $generalized$ suffix tree for strings $LMMNP$ and $LMNMM$ is $LMMNPLMNMM$ The time to build this suffix tree is proportional to the total length of all the strings. The suffix tree for a single string S_1 and a set of string S_1 and S_2 is shown in Fig. 4.2.2.

In a suffix tree structure, we can easily retrieve matching substrings from a set of string by finding the branch that has leaves from the corresponding strings. From our suffix tree example (Fig. 4.2.2b), branches with edge-label M, N, LM, MM, and MN have leaves from both strings S_1 and S_2. Those edge-labels represent the same substring shared by S_1 and S_2.

We construct the suffix tree using Ukkonen's algorithm [5]. To cover every training instance, we build a single $generalized$ suffix tree for all the *Hash String*. The length of the concatenate string is proportional to the sum of all the *Hash String* lengths. Details of Ukkonen's algorithm can be found in [5]. Our implementation requires $O(n \times l)$, where n is the number of instances and l is the maximum length of the *Hash String*.

4.2.3 Frequent Substring Extraction

After constructing the suffix tree, we extract frequent substrings. 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 [6]. The values of $\min_{length}$ and $\min_{support}$ are different for each problem. While fixing the values would sacrifice flexibility, it is also computationally challenging to find optimal values by exhaustive search. We therefore apply local search, a simple yet effective method to provide sufficiently good values in reasonable time.

We use local search to move from the initial values of $\min_{length}$ and $\min_{support}$ to their neighbors by changing either $\min_{length}$ or $\min_{support}$ at each move until the average distance among all instances in two different clusters is no longer improving.

To find the initial values of $\min_{length}$ and $\min_{support}$, we run a competition among 5 candidates, namely: lower bound of $\min_{length}$ and $\min_{support}$, upper bound of $\min_{length}$ and $\min_{support}$, mid value between lower and upper bound and two random values. Lower bound of $\min_{length}$ and $\min_{support}$ is set to 2, while upper bound is set to 20% of the number of instances for $\min_{length}$ and 20% of the maximum string length respectively for $\min_{support}$.

4.2.4 Feature Selection

We use user-specified constraints (*must-link* and *cannot-link constraints*) to select the "*good*" features from the frequent substrings. As in [20], we want to select features with the best constraint preserving ability. We assume that a "*good*" feature should appear more in instances that has *must-link* constraints rather than in instances that has *cannot-link* constraints.

Based on that assumption, we filter the substrings using the following steps:

1. Generate a set of cliques C for instances that belong to the same cluster based on M_{link}
2. Calculate discriminative score $DScore$ for each substring using the following formula:

$$DScore(s) = \frac{\sum_{k=0}^{|C|} \sum_{(i,j) \in C_k} (x(s)_{ij})}{\sum_{(i_i,i_j) \in C_{link}} (x(s)_{ij})} \quad (1)$$

where
$$x(s)_{ij} = \begin{cases} 1 & \text{if } s \text{ is appear in instance } i \text{ and } j \\ 0 & \text{otherwise} \end{cases}$$

3. Select n number of substring which has the highest score as features. The default value for n is $M/2$, where M is the number of substrings.

Note that, if the user-specified constraints is not available, we use all the extracted substrings as features.

4.3 Similarity Score Calculation

After having the features, we calculate the instance's score 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 up the number of repetitions for each symbol in feature from previously constructed *Hash Table*. A frequent substring may occur multiple times in one string. We calculate the score for each occurrence and choose the maximum score as the score for the instance-feature metric.

With this metric, we calculate the similarity for each pair of instances by cosine similarity, a widely-used similarity measure for comparing vectors [5]. Cosine similarity is formulated as:

$$similarity = \frac{\sum_{i=0}^{n} (f_i(I_1) \times f_i(I_2))}{\sqrt{\sum_{i=0}^{n} f_i(I_1)^2} + \sqrt{\sum_{i=0}^{n} f_i(I_2)^2}} \quad (2)$$

where $f_i(I_1)$ and $f_i(I_2)$ are the scores from the instance-feature metric for Instance 1 and 2 respectively.

4.4 Clustering

Similar to [12], we cluster the instances by a well-known clustering approach AGNES [6] with L method [16]. 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 [16] 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.

4.5 Signature Extraction

In testing phase, we use signatures to represent instance's search trajectories in each cluster. We define signature as follows.

Definition 5 (Signature $\mathcal{SIGN}$) *Let c be a cluster of problem instance i, S be a set of search trajectories for all instance i in c, min_{length} be a minimum length and $min_{support}$ be a minimum support. $\mathcal{SIGN}(c)$ is defined as a set of distinct segments from S that has significant length (greater than min_{length}) and appears in most of instance i in c (at least $min_{support}$ number of instances).*

The steps for signature extraction are similar to feature extraction (Section 4.2), but for each cluster, we need to construct a different suffix tree and extract the features from each suffix tree. Since each cluster already satisfy user-specified constraints, we skip the feature selection process and use all the extracted substrings as features. Unless stated otherwise, min_{length} and $min_{support}$ are set to min_{length} and $min_{support}$ value in feature extraction.

4.6 Time Complexity

The time complexity for generating *Hash String* is $O(n \times m)$ with n being the number of instances and m being the maximum string length, while that for constructing the suffix tree is $O(n \times l)$ with n being the number of instances and l being the maximum *Hash String* length. Extracting features, building instance-feature metric and calculating similarity for each pair of instances can be done by a single traversal of the suffix tree structure. Hence it requires $O(n \times l)$ with n being the number of instances and l being the maximum *Hash String* length. The clustering process requires $O(n^2)$ with n being the number of instances. Hence, the overall time complexity for the training phase, excluding the time needed for tuning, is $O(n^2 + (n \times m) + (n \times l))$ with n being the number of instances, m being the maximum string length and l being the maximum *Hash String* length ($n \ll l \ll m$).

In the testing phase, cluster signature construction requires $O(n \times l)$ where n is the number of training instances and l is the maximum *Hash String* length. For a given testing instance, we match the signatures with the testing instance's strings, which takes $O(signature_c \times l_{signature} \times m \times l)$ time, where $signature_c$ is the total number of signatures in all clusters, $l_{signature}$ is the maximum length of signatures, m is the number of testing instances, and l is the maximum string length of the testing instance ($l_{signature} \ll l$).

5 Empirical Evaluation

In this section, we present our experimental results to measure the efficiency and effectiveness of SufTra on a classical COPs, Quadratic Assignment Problem (QAP).We compare our approach with two other instance-specific configurators in the literature: ISAC [10] and CluPaTra [12]. Note that since our aim is to measure solution quality, we do not compare our approach with Hydra [19], another instance-specific configurator that seeks to optimize run time performance but not solution quality of the target algorithm.

As a target algorithm, we use hybrid Simulated Annealing and Tabu Search (SA-TS) algorithm [13], which uses the Greedy Randomized Adaptive Search Procedure (GRASP) to obtain an initial solution, and a combined Simulated Annealing (SA) and Tabu Search (TS) algorithm to improve it. There are four parameters to be tuned as described in Table. 2.

Table 2. Parameters for SA-TS on QAP

Parameter	Description	Type	Range
Temp	Initial temperature of SA algorithm	Continuous	[100, 5000]
Alpha	Cooling factor	Continuous	[0.1, 0.9]
Length	Length of tabu list	Discrete	[1, 10]
Pct	Percentage of non-improving iterations prior to intensification strategy	Continuous	[0.01, 0.1]

We used two set of instances: SET A and SET B as described in Table. 3. All experiments were performed on a 1.7GHz Pentium-4 machine running Windows XP.

Table 3. Set of Instances for QAP

Set	Description	n_{tr}	n_t	m
SET A	benchmark instances from QAPLib with number of city 22 to 150	40	10	4,613
SET B	generated instances from two generators as in [14] with number of facilities varied from 5 to 150	100	400	15,536

5.1 Cluster Analyses

We experimented on SET A instances and compared the clusters created from SufTra, CluPaTra and ISAC. Since ISAC requires problem-specific features, we used 2 features: *flow dominance* and *sparsity* of flow metric [17].

We measured the cluster quality using *extrinsic* method. *Extrinsic* methods compare the clusters against the known class labels or *ground-truth* clusters (i.e. the set of clusters which is supposed to represent the ideal/optimal clustering) [6]. We used the existing well-studied classification of QAP instances based on the distance and flow metrics, due to [18] as *ground truth* cluster.

For this experiment, we used two different SufTra implementation. For SufTra, we did not include user-specified constraints information, while for SufTra(c), we randomly derived 20 user-specified constraints from *ground truth* cluster.

The cluster quality values are shown in Table. 4 (I). Notice that SufTra(c) and SufTra has higher cluster quality compared to CluPaTra and ISAC.

5.2 Performance Comparison

To evaluate SufTra's effectiveness for long search trajectories and large number of instances, we ran experiment on SET B. For SufTra(c), we randomly used 20 user-specified constraints where we derived from the following assumption: (1) instances from the same generator are considered on the same cluster; and (2) instances from different generators are considered on different clusters.

Table 4. QAP Experiment Result

	Training	Testing
I. Clustering Analyses		
CluPaTra	0.68	0.70
ISAC	0.80	0.80
SufTra	0.85	0.85
SufTra(c)	**0.91**	**0.93**
II. Computational Time		
CluPaTra	1,051 s	2,718 s
SufTra	**56 s**	**146 s**
SufTra(c)	65 s	147 s
III. Performance Result		
ParamILS	1.08	2.14
CluPaTra	0.89	1.58
ISAC	0.84	1.22
SufTra	0.83	1.16
SufTra(c)	**0.80**	**1.15**
p-value**	0.061	0.042

**based on statistical test on ISAC and SufTra

First, we compared the time needed (in seconds) for SufTra, SufTra(c) and CluPaTra to form the clusters in training phase and to map the testing instances in testing phase. Table. 4 (II) shows the result. From the table, we observe that SufTra and SufTra(c) is 18 times faster then CluPaTra.

Next, we compared the target algorithm performance using parameter configuration from SufTra, SufTra(c), CluPaTra and ISAC as well as the one-size-fits-all configurator ParamILS. For the three instance-specific methods, we used the same one-size-fits-all configurator, ParamILS [9]. Since ParamILS works only with discrete parameters, we first discretized the values of the parameters. We measured the performance using performance metric as defined in Definition 1.

We set the cutoff runtime of ParamILS to 100 second. For CluPaTra, SufTra and SufTra(c), 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 [6] 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 (III), we show the performance comparison results. From the table, we observe that SufTra and SufTra(c) performs better on training and testing instances compare to other approaches. But the result for training instances is not statistically significant compare to ISAC.

6 Conclusion and Future Works

In this paper, we proposed a new generic instance-based configurator via the clustering of patterns according to the instance search trajectories using the suffix tree data structure and user specified-constraints. We measured the cluster quality and performance of QAP, and show that SufTra result (with or without user-specified constraints) is outperform the existing methods. Its cluster is almost similar to existing benchmark instance classifications, thereby showing its effectiveness. Hence, we claim that: (1) SufTra (with or without user-specified constraints) 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.

Up to this stage of our work, the SufTra framework can only be applied to target algorithms which are local-search-based, since our approach uses search trajectory as the generic feature. As future works, we will investigate how to generate clusters from population-based-algorithm using generic features pertaining to population dynamics.

REFERENCES

[1] Belarmino Adenso-Díaz and Manuel Laguna, 'Fine-tuning of algorithms using fractional experimental design and local search', *Operations Research*, **54**(1), 99–114, (2006).

[2] Carlos Ansótegui, Meinolf Sellmann, and Kevin Tierney, 'A gender-based genetic algorithm for the automatic configuration of algorithms', in *15th international Conference on Principles and Practice of Constraint Programming*, pp. 142–157, (2009).

[3] Sugato Basu, Mikhail Bilenko, and Raymond J. Mooney, 'A probabilistic framework for semi-supervised clustering', in *Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-2004)*, pp. 59–68, (2004).

[4] M. Birattari, Z. Yuan, P. Balaprakash, and T. Stützle, 'Automated algorithm tuning using f-races: Recent development', in *8th Metaheuristics International Conference*, (2009).

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

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

[7] H.H. Hoos and T. Stützle, *Stochastic Local Search: Foundation and Application*, Morgan Kaufman, San Francisco, 2004.

[8] 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).

[9] 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).

[10] S. Kadioglu, Y. Malitsky, M. Sellmann, and K. Tierney, 'Isac:instance-specific algorithm configuration', in *19th European Conference on Artificial Intelligence*, (2010).

[11] G. Klau, N. Lesh, J. Marks, and M. Mitzenmacher, 'Human-guided tabu search', in *In: National Conference on Artificial Intelligence (AAAI)*, (2002).

[12] Lindawati, Hoong Chuin Lau, and David Lo, 'Instance-based parameter tuning via search trajectory similarity clustering', in *LNCS: 5nd Learning and Intelligent OptimizatioN Conference*, (2011).

[13] K.M. Ng, A. Gunawan, and K.L. Poh, 'A hybrid algorithm for the quadratic assignment problem', in *International Conf. on Scientific Computing*, pp. 14–17, (2008).

[14] Gabriela Ochoa, Sebastien Verel, Fabio Daolio, and Marco Tomassini, 'Clustering of local optima in combinatorial fitness landscape', in *LNCS: 5nd Learning and Intelligent OptimizatioN Conference*, (2011).

[15] C.R. Reeves, 'Landscapes, operators and heuristic search', *Annals of Operations Research*, **86**(1), 473–490, (1999).

[16] 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*, pp. 576–584, (2004).

[17] 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).

[18] É.D. Taillard, 'Comparison of iterative searches for the quadratic assignment problem', *Location Science*, **3**(2), 87–105, (1995).

[19] 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).

[20] Daoqiang Zhang, Songcan Chen, and Zhi-Hua Zhou, 'Constraint score: A new filter method for feature selection with pairwise constraints', *Pattern Recognition*, **41**(5), 1440–1451, (2008).