ISAC – Instance-Specific Algorithm Configuration [1]

Serdar Kadioglu and **Yuri Malitsky** and **Meinolf Sellmann** and **Kevin Tierney**[2]

Abstract. We present a new method for instance-specific algorithm configuration (ISAC). It is based on the integration of the algorithm configuration system GGA and the recently proposed stochastic offline programming paradigm. ISAC is provided a solver with categorical, ordinal, and/or continuous parameters, a training benchmark set of input instances for that solver, and an algorithm that computes a feature vector that characterizes any given instance. ISAC then provides high quality parameter settings for any new input instance. Experiments on a variety of different constrained optimization and constraint satisfaction solvers show that automatic algorithm configuration vastly outperforms manual tuning. Moreover, we show that instance-specific tuning frequently leads to significant speed-ups over instance-oblivious configurations.

1 Introduction

When developing a new heuristic or complete algorithm for a constraint satisfaction or a constrained optimization problem, we frequently face the problem of choice. There may be multiple branching heuristics that we can employ, different types of inference mechanisms, various restart strategies, or a multitude of neighborhoods to choose from. Furthermore, the way in which the choices we make affect one another is not readily known. The task of making these choices is known as algorithm configuration.

Developers often make many of the algorithmic choices during the prototyping stage. Based on a few preliminary manual tests, certain algorithmic components are discarded, even before all the remaining components have been implemented. However, by doing this the developers can unknowingly discard algorithmic components that are used in the optimal configuration. In addition, the developer of an algorithm has limited knowledge about the instances that a user will typically employ the solver for. That is the very reason why solvers have parameters; to enable users to fine-tune a solver for their specific needs.

Manually tuning such a solver can take a lot of time and effort. Before even trying the numerous possible parameter settings, the user must learn about the inner workings of the solver to understand what each parameter does. Furthermore, it has even been shown that manual tuning often leads to highly inferior performance [17].

The field of automatic algorithm configuration, which has experienced a renaissance in the past decade, tries to overcome these limitations of manual parameter tuning. The idea is that the developer implements all alternatives of each algorithm component that can be selected via parameters. Then, based on a set of representative problem instances, an automatic configurator tunes the algorithm by selecting the parameters that yield the best performance.

Existing techniques select one parameter set that works reasonably well on all instances in the training set. In this work, we develop a new type of configurator that provides high-quality parameter sets that are based on the specific problem instance that needs to be solved. That is, we make algorithm configuration instance-specific.

2 Related Work

2.1 Automatic Algorithm Configuration

Several approaches exist in the literature for the automatic tuning of algorithms. Some of these were created for a specific algorithm or task. For example, [24] devises a modular algorithm for solving constraint satisfaction problems (CSPs). Using a combination of exhaustive enumeration of all possible configurations and parallel hill-climbing, the technique automatically configures the system for a given set of training instances. Another approach, presented in [30], focuses on the configuration of adaptive algorithms, employing a sequential parameter optimization approach.

Other approaches automatically design and build an entire solver to best tackle a set of example training instances. For example, [28] uses genetic programming to create an evolutionary algorithm (EA). Here the chromosome is an EA operation like the selection of parents, mutation, or crossover, and the task is to find a sequence of the genetic programming operators that is best suited for the specified problem. For SAT, [8] classifies local search (LS) approaches by means of context-free grammars. This approach then uses a genetic programming approach to select a good LS algorithm for a given set of instances.

There also exist approaches that are applicable to more general algorithms. For example, in order to tune continuous parameters, [5] suggests an approach that determines good parameters for individual training instances. This approach first evaluates the extreme parameter configurations and then fits a regression function to map the parameter/performance tuple. The minimization of the resulting function yields a set of parameters for a given instance. For a small set of possible parameter configurations, [3] employs a racing mechanism. During training, all potential algorithms are raced against each other, whereby a statistical test eliminates inferior algorithms before the remaining algorithms are run on the next training instance. Alternatively, the CALIBRA system [1] starts with a factorial design of the parameters. Once these initial parameter sets have been run and evaluated, an intensifying local search routine starts from a promising design, whereby the range of the parameters is limited according to the results of the initial factorial design experiments.

To date, there are only two systems that can configure arbitrary algorithms with very large numbers of parameters, ParamILS proposed by [17], and GGA which was proposed by [2]. ParamILS conducts an iterated local search, whereby a special technique is used to limit the number of training instances that need to be run for each parameter set by focusing the test runs on promising parameter sets. As the name suggests, GGA, an abbreviation for gender-based genetic algorithm, conducts a population-based local search whereby

[1] This work was supported in part by the National Science Foundation through the Career: Cornflower Project (award number 0644113).

[2] Brown University, USA, email: serdark,ynm,sello,ktierney@cs.brown.edu

the separation of a competitive and a non-competitive gender balances exploitation and exploration of the parameter space.

2.2 Algorithm Selection

Algorithm selection is a topic that is closely related to algorithm configuration. Given an instance the objective of the solver is to choose an algorithm that is likely to yield best performance. For example, in [22] a sampling technique selects one of several different branching variable selection heuristics in a branch-and-bound approach.

[11] proposed to run in parallel (or interleaved on a single processor) multiple stochastic solvers that tackle the same problem. These "algorithm portfolios" were shown to work much more robustly than any of the individual stochastic solvers. This insight has since then led to the technique of randomization with restarts which is commonly used in all state-of-the-art complete SAT solvers.

On the other hand, [18] suggested to use algorithm portfolios in a different way. Like before, they consider multiple algorithms for the same problem. However, in this approach, a forecast of the runtime is made for each algorithm for any given input instance based on characteristic instance features. Then the algorithm with the shortest predicted runtime is employed. SATzilla [35] is a prominent example of this approach for SAT. Since its initial introduction in 2007, SATzilla has consistently been ranked highly at the SAT Competitions [31].

2.3 Instance-Specific Tuning

What is interesting about algorithm selection is that it considers the input instance when "configuring" the solver to pick the correct algorithm. The limitation of these methods is, of course, that the given portfolio can only consist of a small set of solvers. Compared to algorithm configuration, the number of choices in these portfolios is extremely limited. Therefore, when an entire family of algorithms, as represented by exponentially or even infinitely many parameter settings is given, it is no longer possible to learn a prediction model for each different setting.

Some methods therefore try to integrate the benefits of both approaches, considering a parameterized solver, i.e. an entire family of algorithms in the portfolio, and base the selection of parameters instance-specifically according to the features of the input instance. In [29], a self-tuning WalkSAT approach is presented that chooses WalkSAT parameters based on the input instance. In another approach, [15, 16] tackle solvers with continuous and ordinal (but not categorical) parameters. Here, Bayesian linear regression is used to learn a mapping from features and parameters into a prediction of runtime. Based on this mapping for given instance features, a parameter set which minimizes predicted runtime is searched for. The approach in [15] led to a twofold speed-up for the local search SAT solver SAPS.

Alternatively, instead of using regression to map instance features to a parameter configuration, [23] introduced the stochastic offline programming paradigm. This is an iterated three step approach. First, the training instances are clustered into distinct sets based on the similarity of their feature vectors. Then, assuming that instances with alike features behave similarly under the same algorithm, local search is used to find good parameters for each cluster of instances. Finally, the algorithm refines the distance metric in the feature space so that it can find better clusters in future iterations. The entire procedure is repeated until no significant improvement or changes are achieved. Experiments on the set covering problem showed that the solutions computed by a randomized greedy algorithm can be massively improved in this way. The same paper also showed that, in this domain, regression-based learning of instance-specific parameters leads to no improvement over the best instance-oblivious parameters.

3 Instance-Specific Algorithm Configuration

The objective of this work is to develop a general configurator that can tune any solver and choose solver parameters according to the instance to be solved. Based on the limited research that has been conducted on this subject, we decided to continue the most successful approach so far, stochastic offline programming, which is based on the clustering of instances followed by the computation of high-quality parameters for all instances within each cluster.

Clustering is advantageous for parameter tuning for several reasons. First, training parameters on a collection of instances generally provides more robust parameters than one could obtain when tuning on individual instances. That is, tuning on a collection of instances helps prevent over-tuning and allows parameters to generalize to similar instances. Secondly, the found parameters are "pre-stabilized," meaning they are proven to work well *together*. Note that this is not the case for the approaches presented in [15, 16], which may provide parameter sets that have never before been run in combination.

In order to use clustering, a metric in the feature space must be provided. To this end, the approach in [23] employs the loss in performance when using a parameter set computed for one cluster on an instance from another. This works well when improving solution quality of a heuristic for set covering, where it is possible to perfectly assess algorithm performance. The situation changes when our objective is to minimize runtime. This is because parameter sets that are not well suited for an instance are likely to run for a very long time, necessitating the need to introduce a timeout. This then implies that we do not always know the real performance, and all we can use is a lower bound on the desired distance between two points in the feature space.

This complicates learning a new metric for the feature space. In our experiments, for example, we found that most instances from one cluster timed out when run with the parameters of another. This not only leads to poor feature metrics, but also costs a lot of processing time. Consequently, for the purpose of tuning the speed of general solvers we suggest a different approach. Instead of learning a feature metric over several iterations, we normalize the features using translation and scaling so that, over the set of training instances, each feature spans exactly the interval $[-1, 1]$. That is, for each feature there exists at least one instance for which this feature has value 1 and at least one instance where the feature value is -1 (whereby we discard features which are identical for all training instances). For all other instances, the value lies between these two extremes.

Another issue with the approach from [23] is that it employs k-means for clustering. This algorithm first selects k random points in the feature space. It then alternates between two steps until some termination criterion is reached. The first step assigns each instance to a cluster according to the shortest distance to one of the k points that were chosen. The next step then updates the k points to the centers of the current clusters.

The problem with k-means clustering is that it requires the user to explicitly specify the number of clusters k. If k is too low, this means that we lose some of our potential to tune parameters more precisely for different parts of the instance feature space. On the other hand, if there are too many clusters, we sacrifice the robustness and generality of the parameter sets that we optimize for these clusters. Furthermore, for a mixed set of training instances, it is unreasonable to assume that the value of k is known.

```
1:  $g$-Means($X$)
2:  $k \leftarrow 1, i \leftarrow 1$
3:  $(C, S) \leftarrow k$-Means$(X, k)$
4:  while $i \leq k$ do
5:      $(\bar{C}, \bar{S}) \leftarrow k$-Means$(S_i, 2)$
6:      $v \leftarrow \bar{C}_1 - \bar{C}_2, w \leftarrow \sum v_i^2$
7:      $y_i \leftarrow \sum v_i x_i / w$
8:      if Anderson-Darling-Test($y$) failed then
9:          $C_i \leftarrow \bar{C}_1, S_i \leftarrow \bar{S}_1$
10:         $k \leftarrow k + 1$
11:         $C_k \leftarrow \bar{C}_2, S_k \leftarrow \bar{S}_2$
12:     else
13:         $i \leftarrow i + 1$
14:     end if
15: end while
16: return (C, S, k)
```

Algorithm 1: g-means Clustering Algorithm

We address this issue by using g-means, a clustering algorithm proposed in [12] which automatically determines the number of clusters. [12] proposes that a good cluster exhibits a Gaussian distribution around the cluster center. The algorithm, presented in Algorithm 1, first considers all inputs as forming one large cluster. In each iteration, we pick one of the current clusters and try to assess whether it is already sufficiently Gaussian. To this end, g-means splits the cluster in two by running 2-means clustering. We can then project all points in the cluster onto the line that runs through the centers of the two sub-clusters, obtaining a one-dimensional distribution of points. g-means now checks whether this distribution is normal using the widely accepted statistical Anderson-Darling test. If the current cluster does not pass the test, it is split into the two previously computed clusters, and we continue with the next cluster.

We found that the g-means algorithm works very well for our purposes. The only problem we encountered was that sometimes clusters can be very small, containing very few instances. To obtain robust parameter sets we do not allow clusters that contain fewer than a manually chosen threshold, a value which depends on the size of the data set. Beginning with the smallest cluster, we re-distribute the corresponding instances to the nearest clusters, where proximity is measured by the Euclidean distance of each instance to the cluster's center.

In summary, our approach works as follows (see Algorithm 2). In the learning phase, we are provided with the parameterized solver A, a list of training instances T, and their corresponding feature vectors F. First, we normalize the features in the set and memorize the scaling and translation values for each feature (s, t).

Then, we use the g-means algorithm to cluster the training instances based on the normalized feature vectors. The resulting small clusters with too few instances are re-distributed to larger clusters as discussed above. The final result of the clustering is a number of k clusters S_i, a list of cluster centers C_i, and, for each cluster, a distance threshold d_i which determines when a new instance will be considered as close enough to the cluster center to be solved with the parameters computed for instances in this cluster.

Then, for each cluster of instances S_i we compute favorable parameters P_i using the instance-oblivious tuning algorithm GGA. After this is done, we compute parameter set R for all the training instances. This serves as the recourse for all future instances that are not near any of the clusters.

We use GGA because it is one of the most competitive and robust tuners available, able to handle any type of parameter. Also, [2] compared GGA to ParamILS, the only other viable option, and

```
1: ISAC-Learn($A, T, F$)
2: $(\bar{F}, s, t) \leftarrow$ Normalize$(F)$
3: $(k, C, S, d) \leftarrow$ Cluster $(T, \bar{F})$
4: for all $i = 1, \ldots, k$ do
5:     $P_i \leftarrow GGA(A, S_i)$
6: end for
7: $R \leftarrow GGA(A, T)$
8: return $(k, P, C, d, s, t, R)$

1: ISAC-Run($A, x, k, P, C, d, s, t, R$)
2: $f \leftarrow$ Features$(x)$
3: $\bar{f}_i \leftarrow (f_i - t_i)/s_i \; \forall \; i$
4: for all $j = 1, \ldots, k$ do
5:     if $||\bar{f} - C_i|| \leq d_i$ then
6:         return $A(x, P_i)$
7:     end if
8: end for
9: return $A(x, R)$
```

Algorithm 2: Instance-Specific Algorithm Configuration

showed significant gains in performance and robustness for GGA over ParamILS.

When running algorithm A on an input instance x, we first compute the features of the input and normalize them using the previously stored scaling and translation values for each feature. Then, we determine whether there is a cluster such that the normalized feature vector of the input is close enough to the cluster center. If so, we run A on x using the parameters for this cluster. If the input is not near enough to any of our clusters we use the instance-oblivious parameters R that work well for the entire training set. Specifically, in our experiments, an instance was considered too far away if it was more than the average distance plus two standard deviations of the distance of all points in the cluster to its center. This tended to be only 9% of the test instances.

4 Numerical Results

4.1 Set Covering

We begin our empirical evaluation on one of the most studied combinatorial optimization problems: the set covering problem (SCP). In SCP, given a finite set $S := \{1, \ldots, m\}$ of items, a family $F := \{S_1, \ldots, S_n \subseteq S\}$ of subsets of S, and a cost function $c : F \rightarrow \mathbb{R}^+$, the objective is to find a subset $C \subseteq F$ such that $S \subseteq \bigcup_{S_i \in C} S_i$ and $\sum_{S_i \in C} c(S_i)$ is minimized. In the *unicost* SCP, the cost of each set is set to one. This problem formulation appears in numerous practical applications such as crew scheduling [14, 13, 4], location of emergency facilities [33], and production planning in various industries [34].

Solvers: We consider three different solvers. The first is the greedy randomized set covering heuristic "GS" from [23]. GS repeatedly adds sets one at a time until reaching a feasible solution. During the construction of the cover, a probability distribution is used to specify the set selection heuristic at each step. The other two solvers are state-of-the-art local search SCP solvers. The dialectic search algorithm "Hegel" was introduced in [20], and the tabu search algorithm "TS" was introduced in [27] which is restricted to unicost instances.

Benchmark: A highly diverse set of randomly generated set covering instances was introduced in [23]. These instances involve up to 100 items and 10,000 sets. We pre-compute the optimal values of these instances. When tuning TS, we set the cost of each set uniformly to 1 to achieve unicost instances. The final data set comprises

200 training instances and 200 test instances.

Instance Features: The generation of a feature vector for each SCP was done according to the process outlined in [23]. This process first computes the following:

- normalized cost vector $c' \in [1, 100]^m$,
- vector of bag densities $(|S_i|/n)_{i=1\dots m}$,
- vector of item costs $(\sum_{i,j \in S_i} c'_i)_{j=1\dots n}$,
- vector of item coverings $(|\{i \mid j \in S_i\}|/m)_{j=1\dots n}$,
- vector of costs over density $(c'_i/|S_i|)_{i=1\dots m}$,
- vector of costs over square density $(c'_i/|S_i|^2)_{i=1\dots m}$,
- vector of costs over $k \log k$-density $(\frac{c'_i}{(|S_i| \log |S_i|)})_{i=1\dots m}$, and
- vector of root-costs over square density $(\sqrt{c'_i}/|S_i|^2)_{i=1\dots m}$.

The final feature vector is then formed by computing the maxima, minima, averages, and standard deviations of all these vectors. Computation of these feature values on average took only 0.01 seconds per instance.

Numerical Results: Unless otherwise noted, experiments were run on dual processor dual core Intel Xeon 2.8 GHz computers with 8GB of RAM. SCP solvers Hegel and TS were evaluated on quad core dual processor Intel Xeon 2.53 Ghz processors with 24GB of RAM.

The first objective of our experiments is to compare ISAC with the stochastic offline programming (SOP) approach from [23]. When deterministically using the single best greedy heuristic, GS leaves, on average, a 7.2% (7.6%) optimality gap on the training (testing) data [23]. It is possible to shrink this gap using a uniform distribution over the six set selection heuristics used in GS. We refer to this default version of the GS as Uniform. In [23], a configurator was developed which could compute high-quality instance-oblivious GS parameters. We refer to this approach as SOP-combined. Moreover, [23] showed that clustering the training data into sets, and then tuning these sets individually could lead to further improvements. We refer to this approach as SOP-clustered. We compare these two configurators with general-purpose instance-oblivious configuration of GGA [2] and instance-specific parameter tuning of ISAC.

Solver: GS	% Optimality Gap Closed	
	Train	Test
Uniform	25.9 (4.2)	40 (4.1)
SOP - combined	39.0 (3.3)	43.4 (3.6)
SOP - clustered	**47.7 (2.4)**	50.3 (3.7)
GGA	40.0 (3.6)	46.1 (3.8)
ISAC	44.4 (3.3)	**51.3 (3.8)**

Table 1. Comparison of ISAC versus the default and instance-oblivious parameters provided by SOP and GGA, and the instance-specific parameters provided by SOP. We present the percent of optimality gap closed (stdev).

In Table 1, we compare the resulting five GS solvers, presenting the percentage of optimality gap closed by each solver. Comparing the average percent of optimality gap closed, we find that ISAC is as capable of improving over the default approach as SOP, which was developed particularly for the GS solver. That is, ISAC can effectively liberate us from having to select the number of clusters while, at the same time, enjoying wide applicability to other solvers. Moreover, ISAC works more efficiently than SOP since it does not require multiple re-clustering steps for learning a metric in the instance feature space.

Table 1 shows a clear distinction between the instance-specific and the instance-oblivious tuning methods. Both ISAC and SOP-clustered perform significantly better than GGA and SOP-combined. Instance-specific tuning is best realized by ISAC, closing the optimality gap by more than 50% on average.

We next evaluate ISAC on two state-of-the-art local search SCP solvers; Hegel and TS. For both solvers we measure the time to find a set covering solution that is within 10% of optimal. Hegel and TS had a timeout of 10 seconds for training and testing.

In Table 2, we compare the default configuration of the solvers, the instance-oblivious configuration obtained by GGA, and the instance-specifically tuned versions provided by ISAC. To provide a more holistic view of ISAC's performance, we present three performance metrics: the arithmetic and geometric means of the runtime in seconds and the average slow down (the arithmetic mean of the ratios of the performance of the competing solver over ISAC).

For these experiments we set the minimum cluster size to 30 instances. This setting resulted in 4 clusters of roughly equal size.

Solver		Avg. Run Time		Geo. Avg.		Avg. Slow Down	
		Train	Test	Train	Test	Train	Test
TS	Default	2.79	3.45	2.36	2.60	1.49	1.79
	GGA	2.58	3.40	2.27	2.63	1.35	1.72
	ISAC	**1.99**	**2.04**	**1.96**	**1.97**	**1.00**	**1.00**
Hegel	Default	3.04	3.15	2.52	2.49	2.20	2.03
	GGA	1.58	1.95	**1.23**	**1.33**	1.10	1.15
	ISAC	**1.45**	**1.92**	**1.23**	1.36	**1.00**	**1.00**

Table 2. Comparison of default, instance-oblivious parameters provided by GGA, and instance-specific parameters provided by SOP for Hegel and TS. We present the arithmetic and geometric mean runtimes in seconds and the average degradation when comparing each solver to ISAC.

We first observe that the default configuration of both solvers can be improved significantly by automatic parameter tuning. For solver TS, we measure an arithmetic mean runtime of 2.18 seconds for ISAC-TS, 3.33 seconds for GGA-TS, and 3.44 seconds for default TS. That is, instance-oblivious parameters run 50% slower than instance-specific parameters. For Hegel, we find that the default version runs more than 60% slower than ISAC-Hegel.

It is worth noting that we observe a high variance of the runtimes from one instance to another, which is caused by the diversity of our benchmark. To get a better understanding, we also compute the average slow down of each solver when compared with the corresponding ISAC version. For this measure we find that, for an average test instance, default TS requires more than 1.70 times the time of ISAC-TS, and GGA-TS needs 1.62 times over ISAC-TS. For default Hegel, an average test instance takes 2.10 times the time of ISAC-Hegel while GGA-Hegel only runs 10% slower. This confirms the findings in [20] that Hegel runs robustly over different instance classes with one set of parameters.

We conclude that even highly sophisticated, state-of-the-art solvers can greatly benefit from automatic parameter tuning. Depending on the solver, instance-specific parameter tuning works as well or significantly better than instance-oblivious tuning. Note that this is not self-evident since the instance-specific approach runs the risk of over-tuning by considering fewer instances per cluster. In our experiments, we do not observe these problems. Instead we find that our instance-specific algorithm configurator offers the potential for great performance gains without over-fitting the training data.

4.2 Mixed Integer Programming

We next consider mixed integer programming problems (MIPs). MIPs involve optimizing a linear objective function while obeying a collection of linear inequalities and variable integrality constraints. Mixed integer programming is an area of great importance in operations research.

Solver: The fastest and best known MIP solver is IBM Cplex [6]. For 15 years it has represented the state-of-the-art in MIP solving. The solver is ideal for our purposes because it is highly parameterized, allowing the user to precisely choose the settings they think are best suited for their MIP instances. For these experiments we use Cplex 12.1.

Benchmark: We assembled a highly diverse benchmark data set composed of problem instances from six different sources. Network flow instances, capacitated facility location instances, bounded and unbounded mixed knapsack instances and capacitated lot sizing problems, all taken from [25], as well as combinatorial auction instances from [21]. In total there are 588 instances in this set, which was split into 276 training and 312 test instances.

Instance Features: Even though solving MIPs is an active field, to the best of our knowledge no prior research exists on the type of features that can be used to classify a MIP instance. We therefore propose to use the information about the objective vector, the right hand side (RHS) vector, and the constraint matrix to formulate our feature vector. We first compute the following values:

- number of variables and number of constraints,
- percentage of binary (integer or continuous) variables,
- percentage of variables (all, integer, or continuous) with non zero coefficients in the objective function, and
- percentage of $\leq$ ($\geq$ or $=$) constraints.

We also use the mean, min, max, and standard deviation of the following vectors, where $Z = \{x_i \mid x_i \text{ is restricted to be integer}\}$, $R = \{x_i \mid x_i \text{ is real valued}\}$, and $U = Z \cup R$:

- vector of coefficients of the objective function (of all, integer, or continuous variables): $(c_i | x_i \in X)$ where $X = U \vee X = Z \vee X = R$,
- vector of RHS of the $\leq$ ($\geq$ or $=$) constraints: $(b_j | A_j x \circ b_j)$ where $\circ = (\geq) \vee \circ = (\leq) \vee \circ = (=)$,
- vector of number of variables (all, integer or continuous) per constraint j: $(\#\{A_{(i,j)} \mid A_{(i,j)} \neq 0, x_i \in X\})$ where $X = U \vee X = Z \vee X = R$,
- vector of the coefficients of variables (all, integer, or continuous) per constraint j: $(\sum_i A_{(i,j)} | \forall j, x_i \in X)$ where $X = U \vee X = Z \vee X = R$, and
- vector of the number of constraints each variable i (all, integer, or continuous) belongs to: $(\#\{A_{(i,j)} \mid A_{(i,j)} \neq 0, x_i \in X\}$ where $X = U \vee X = Z \vee X = R$.

Computation of these feature values on average took only 0.02 seconds per instance.

Numerical Results: Experiments were carried out with a timeout of 30 seconds for training and 300 seconds for evaluation on the training and testing sets. We set the size of the smallest cluster to be 30 instances. This resulted in 5 clusters, where 4 consisted of only one problem type, and 1 cluster combined network flow and capacitated lot sizing instances.

Solver		Avg. Run Time		Geo. Avg.		Avg. Slow Down	
		Train	Test	Train	Test	Train	Test
Cplex	Default	6.1	7.3	2.5	2.5	2.0	1.9
	GGA	3.6	5.2	1.7	1.8	1.3	1.2
	ISAC	**2.9**	**3.4**	**1.5**	**1.6**	**1.0**	**1.0**

Table 3. Comparison of ISAC versus the default and the instance-oblivious parameters provided by GGA when tuning Cplex. We present the arithmetic and geometric mean runtimes as well as the average slowdown per instance.

Table 3 compares instance-specific ISAC with instance-oblivious GGA and the default settings of Cplex. We observe again that the default parameters can be significantly improved by tuning the algorithm for a representative benchmark. On average, ISAC-Cplex needs 3.4 seconds, GGA-Cplex needs 5.2 seconds and default Cplex requires 7.3 seconds. Instance-obliviously tuned Cplex is 50% slower, and default Cplex even more than 110% slower than ISAC-Cplex.

The improvements achieved by automatic parameter tuning can also be seen when considering the average per-instance slow-downs. According to this measure, for a randomly chosen instance in the test set we expect that GGA-Cplex needs 20% more time than required by ISAC-Cplex. Default Cplex even needs 90% more time than ISAC-Cplex.

We would like to note that due to license restrictions we could only use a very small training set of 276 instances which is very few given the high diversity of the considered benchmark. Taking this into account and seeing that Cplex is a highly sophisticated and extremely well-tuned solver, the fact that ISAC boosts performance so significantly is surprising and shows the great potential of instance-specific tuning.

4.3 Satisfiability

Our final evaluation of ISAC is on the propositional satisfiability problem (SAT), the prototypical NP-complete problem that has far reaching effects on many areas of computer science. For SAT, given a propositional logic formula F in conjunctive normal form, the objective is to determine whether there exists a satisfying truth assignment to the variables of F. In recent years, there has been tremendous progress in solving SAT problems, so that state-of-the-art SAT solvers can now tackle instances with hundreds of thousands of variables and over one million clauses.

Solvers: We test ISAC on the highly parameterized stochastic local search solver SAPS [19]. Unlike most existing SAT solvers, SAPS was originally designed with automatic tuning in mind and therefore all of the parameters influencing the solver are readily accessible to users. Furthermore, since it was first released, the default parameters of the solver have been drastically improved by general purpose parameter tuners [2, 17].

Benchmarks: We consider the collection of SAT instances described in Table 4. QCP is a collection of quasi-group completion problems, SWGCP contains small-world based graph coloring problems, 3SAT-random has 3SAT instances created using the G2 generator, and 3SAT-structured are instances that are modeled to mimic real world search problems.

Data set	Train	Test	Ref.
QCP	1000	1000	[10]
SWGCP	1000	1000	[9]
3SAT-random	800	800	[26]
3SAT-structured	1000	1000	[32]

Table 4. Data sets used to evaluate ISAC on SAT.

Instance Features: We utilize the features proposed by [35] to classify each problem instance. We find that while the local search features mentioned in [35] take a considerable amount of time to compute, they are not imperative to finding a good clustering of instances. Consequently, we exclude them and use only the following:

- problem size features: number of clauses c, number of variables v, and their ratio c/v,
- variable-clause graph features: degree statistics for variable and clause nodes,
- variable graph features: node degree statistics,
- balance features: ratio of positive to negative literals per clause, ratio of positive to negative occurrences of each variable, fraction of binary and ternary clauses,
- proximity to horn clauses: fraction of horn clauses and statistics on the number of occurrences in a horn clause for each variable,

- unit propagations at depths 1, 4, 16, 64 and 256 on a random path in the DPLL [7] search tree, and
- search space size estimate: mean depth to contradiction and estimate of the log of number of nodes.

Computation of these feature values on average took only 0.01 seconds per instance.

Numerical Results: Experiments were carried out with a timeout of 30 seconds for training and 300 seconds for evaluation on the training and testing sets. We set the size of the smallest cluster to be at least 100 instances. This resulted in 18 clusters each with roughly 210 instances. Here not only were all of the 4 types of instances correctly separated into distinct clusters, a further partition of instances from the same class was provided.

We evaluate the performance of SAPS using the default parameters, GGA, and ISAC and present the results in Table 5.

Solver		Avg. Run Time		Geo. Avg.		Avg. Slow Down	
		Train	Test	Train	Test	Train	Test
SAPS	Default	79.7	77.4	0.9	0.9	292.5	274.1
	GGA	14.6	14.6	0.2	0.2	5.5	4.7
	ISAC	**4.0**	**5.0**	**0.1**	**0.1**	**1.0**	**1.0**

Table 5. Comparison of the SAPS solvers with default, GGA tuned, and ISAC-SAPS. The arithmetic and geometric mean runtimes in seconds are presented as well as the average slow-down per instance.

Even though the default parameters of SAPS have been tuned heavily in the past [17], tuning with GGA solves the benchmark over 5 times faster than default SAPS. Instance-specific tuning allows us to gain another factor of 2.9 over the instance-oblivious parameters, resulting in a total performance improvement of over one order of magnitude. This refutes the conjecture of [16] that SAPS may not be a good solver for instance-specific parameter tuning.

It is worth noting that over 95% of instances in this benchmark can be solved in under 15 seconds. Consequently, some exceptionally hard, long-running instances greatly dilute the average runtime. We therefore present again the average slow-down per instance. For the average SAT instance in our test set, default SAPS runs 274 times slower than ISAC. Even if we use GGA to tune a parameter set specifically for this benchmark, GGA is still expected to run almost 5 times slower than ISAC.

5 Conclusion

In this paper we presented ISAC, a new automatic algorithm configurator that provides high-quality parameter sets based on instance-specific information. The proposed approach has two major steps. First, employing normalized features the training instances are clustered using g-means, a clustering algorithm that automatically determines the appropriate number of clusters. We assume that instances with similar features are likely to behave similarly when solved by the same solver. Therefore, the second step uses the instance-oblivious algorithm configurator GGA to find the best parameters for each cluster. At runtime, when a new instance needs to be solved, we determine the cluster that is closest to the input instance feature vector and then solve the instance with the parameters for the respective cluster. For instances that are very far from all clusters we use instance-oblivious parameters obtained by GGA on the entire set of training instances.

To our knowledge, ISAC is the first instance-specific configuration algorithm that can handle a large number and any type of parameters: continuous, ordinal, as well as categorical. We evaluated ISAC on five high-performance solvers for three different problems: set covering, mixed integer programming, and satisfiability. In all cases we found that automatic algorithm configuration could boost the performance of the default solvers, including the cutting edge solvers Hegel for set covering and Cplex for mixed integer programming. Moreover, we found that instance-specific tuning never works worse than instance-oblivious configuration. On the contrary, ISAC outperformed the instance oblivious tuner GGA in all cases, for most solvers quite substantially.

REFERENCES

[1] B. Adenso-Diaz and M. Laguna. Fine-tuning of Algorithms using Fractional Experimental Design and Local Search. *Operations Research*, 54(1):99–114, 2006.

[2] C. A. Gil, M. Sellmann and K. Tierney. A Gender-Based Genetic Algorithm for the Automatic Configuration of Algorithms. *CP 2009, Springer LNCS 5732*, pp. 142–157, 2009.

[3] M. Birattari, T. Stuetzle, L. Paquete and K. Varrentrapp, A Racing Algorithm for Configuring Metaheuristics. *GECCO*, 11–18, 2002.

[4] A. Caprara, M. Fischetti, P. Toth, D. Vigo and P.L. Guida. Algorithms for Railway Crew Management. *Mathematical Programming*, 79:125–141, 1997.

[5] S.P. Coy, B.L. Golden, G.C. Runger and E.A. Wasil. Using Experimental Design to Find Effective Parameter Settings for Heuristics. *Journal of Heuristics*, 7(1):77–97, 2001.

[6] IBM. *IBM CPLEX* Reference manual and user manual. V12.1, IBM 2009.

[7] M. Davis, G. Logemann, D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7), 394–397, 1962.

[8] A. Fukunaga Automated discovery of local search heuristics for satisfiability testing. *Evolutionary Computation*, 16(1):31–61, 2008.

[9] I. P. Gent, H. H. Hoos, P. Prosser and T. Walsh. Morphing: Combining Structure and Randomness. *AAAI 99*, 654–660, 1999.

[10] C.P. Gomes and B. Selman. Problem Structure in the Presence of Perturbations. *AAAI 97*, 221–226, 1997.

[11] C.P. Gomes, B. Selman. Algorithm Portfolios. *Artificial Intelligence*, 126(1–2):43–62, 2001.

[12] G. Hamerly and C. Elkan. Learning the K in K-Means. *NIPS* 2003.

[13] E. Housos and T. Elmoth. Automatic Optimization of Subproblems in Scheduling Airline Crews. *Interfaces*, 27(5):68–77, 1997.

[14] K.L. Hoffmann and M.W. Padberg. Solving Airline Crew Scheduling Problems by Branch-and-Cut. *Management Science*, 39(6):657–682, 1993.

[15] F. Hutter and Y. Hamadi. Parameter Adjustment Based on Performance Prediction: Towards an Instance-Aware Problem Solver. Technical Report MSR-TR-2005-125, Microsoft Research Cambridge, UK, 2005.

[16] F. Hutter, Y. Hamadi, H.H. Hoos and K. Leyton-brown. Performance prediction and automated tuning of randomized and parametric algorithms. *CP 06*, pp. 213–228, 2006.

[17] F. Hutter, H.H. Hoos, K. Leyton-Brown and T. Stuetzle. ParamILS: An Automatic Algorithm Configuration Framework. *JAIR* Volume 36, pages 267–306, 2009.

[18] B. Huberman, R. Lukose and T. Hogg. An Economics Approach to Hard Computational Problems. *Science*, 265:51–54, 2003.

[19] F. Hutter, D.A.D. Tompkins and H.H. Hoos. Scaling and probabilistic smoothing: Efficient dynamic local search for SAT. *CP 02*, 233–248, 2002.

[20] S. Kadioglu and M. Sellmann. Dialectic Search. *CP 09, Springer LNCS 5732*, 486–500, 2009.

[21] K. Leyton-Brown, M. Pearson and Y. Shoham. Towards a universal test suite for combinatorial auction algorithms. *Proceedings of the ACM Conference on Electronic Commerce (ACM-EC)*, pp. 66–76, 2000.

[22] L. Lobjois and M. Lemaître. Branch and bound algorithm selection by performance prediction. *AAAI 98*, pp. 353–358, 1998.

[23] Y. Malitsky and M. Sellmann. Stochastic Offline Programming. *ICTAI* 784–791, 2009.

[24] S. Minton. Automatically Configuring Constraint Satisfaction Programs. *Constraints*, 1(1):1–40, 1996.

[25] A. Saxena. MIP Benchmark Instances. `http://www.andrew.cmu.edu/user/anureets/mpsInstances.htm`.

[26] M. Motoki and R. Uehara. Unique solution instance generation for the 3-Satisfiability (3SAT) problem.In SAT 2000, pages 293–305.

[27] N. Musliu. Local Search Algorithm for Unicost Set Covering Problem. *IEA/AIE*, 302–311, 2006.

[28] M. Oltean. Evolving evolutionary algorithms using linear genetic programming. *Evolutionary Computation*, 13(3):387–410, 2005.

[29] D. J. Patterson and H. Kautz. Auto-walksat: a self-tuning implementation of walksat. *Electronic Notes in Discrete Mathematics*, Volume 9, pp. 360–368, 2001.

[30] M. Preuss and T. Bartz-Beielstein. Sequential Parameter Optimization Applied to Self-adaptation for Binary-coded Evolutionary Algorithms. *Parameter Setting in Evolutionary Algorithms: Studies in Computational Intelligence*, 91–119, 2007.

[31] SAT Competition. `http://www.satcomptition.org`.

[32] A. Slater. Modelling More Realistic SAT Problems. *15th Australian Joint Conference on Artificial Intelligence, LNAI 2557*, pp. 291–602 2002.

[33] C. Toregas, R. Swain, C. ReVelle and L. Bergman. The Location of Emergency Service Facilities. *Operational Research*, 19(6):1363–1373, 1971.

[34] F.J. Vasko and F.E. Wolf. Optimal Selection of Ingot Sizes via Set Covering. *Operations Research*, 35:115–121, 1987.

[35] L. Xu, F. Hutter, H. H. Hoos and K. Leyton-Brown. SATzilla: Portfolio-based Algorithm Selection for SAT. *JAIR*, Volume 32, pages 565-606, 2008.