Automatic Instance Clustering for Parallel Algorithm Configurators

Frank Mugrauer, Juri Schulte

University Ulm
Institute of Theoretical Computer Science
89069 Ulm, Germany
`frank.mugrauer@uni-ulm.de, juri.schulte@uni-ulm.de`

Abstract. Automatic algorithm configuration has become a promising way to improve the efficiency of heuristic algorithms. The tuning performance depends heavily on the quality of the set of training instances available to the configurator. In this paper, we present a way to analyse and categorise the given training instances, and then exploit this information to improve our configurator. We evaluate this approach in several experiments.

1 Introduction

Many of today's state of the art algorithms have an - at times rather large - number of parameters which allow the algorithm to be tailored to the specific problems that need to be solved. For heuristic algorithms in particular, choosing good values for these parameters can drastically impact the performance of the algorithm. However, deciding which values are the right ones is a very difficult and time consuming process. To mend this problem, a number of automatic *configurators* (e.g. [4]) have been developed. These configurators run the target algorithm with different parameter *configurations* on a set of problem instances, successively trying to improve performance by finding better configurations.

A configurator generally consists of two parts: A *search* and a *racing* procedure. The search procedure's task is to find new promising configurations for future evaluation. The algorithm used in this paper is based on iterative local search (ILS) ([4]), and works by successively evaluating the *neighbourhood* of the best known configuration (or *incumbent*). A configuration is in the incumbent's neighbourhood, if they are different from each other in precisely one parameter value. If the neighbourhood contains a better configuration than the incumbent, this configuration becomes the new incumbent. This approach has been parallelised in [9] and [8].

To decide whether one configuration is to be considered better than another one, a racing procedure is needed. It determines how many runs on problem instances a configuration gets. The more runs a configuration has, the more accurate, meaningful and expensive its evaluation is. One racing procedure that fits well with ILS is ROAR [5], which assigns the new configuration a small number of instances to begin with, and - if it is able to beat the incumbent on those - successively assigns it more instances. If it can beat the incumbent on all the instances that the incumbent has completed, it becomes the new incumbent.

Empiric results show that the selection of instances - especially the very first ones on which a configuration is evaluated - substantially impact the performance of the configurator. If these first instances happen to be very homogeneous, a configuration that performs poorly on this specific subset of instances, but very well on most other instances, will be prematurely discarded by ROAR. This leads to the hypothesis that configurators could be improved by analysing and then carefully selecting instances instead of choosing them randomly or based on a predetermined course. We therefore develop a way to select instances based on their properties, and include this, as well as our implementation of parallelised and adapted versions of ILS and ROAR in the AAC framework for automated algorithm configuration.

2 Technical Framework

This section gives an overview over the software, frameworks and resources used for implementation and experiments.

AAC AAC [1] is a framework for creating generic racing and search procedures implemented in Java. It strives to provide interoperability so that every racing procedure can be used with every search procedure. Basic functionality such as assigning evaluation runs and assessing the results based on a cost function are already provided. It relies on an interface to EDACC.

EDACC EDACC [1] is a software that supports planning and executing algorithm experiments. It has functionality for both manual and automatic algorithm configuration and analysis, aswell as a client program able to use the computational power of the BW-GRID to facilitate algorithm runs.

BW-GRID The BW-GRID [2] is a distributed computing cluster set up by eight universities in Baden-Wuertemberg. It provides computational power for research and educational purposes.

R One of the clustering algorithms used in this paper is provided by the statistics programming language R ([3]).

3 Instance Analysing

Our implementation of ILS and ROAR is based on the versions described in [9] and [8] and therefore includes all of their features (e.g. parallelisation, multiple neighbourhood search, adaptive capping). In addition to this, the size of

[1] https://github.com/ceari/aac

ILS' neighbourhoods is automatically adjusted based on how well the configurations therein are currently performing. On this baseline, we develop our instance analysis. It consists of the following parts:

1. Resource Base
2. Clustering Algorithm
3. Instance Prioritisation

There is a variety of data that can be used as a *resource base* for instance clustering. We provide implementations for two possible resources: *Average instance cost* and *instance properties* as described in [10]. Average instance cost is based on the result of AAC's cost function that evaluates the performance of a configuration on one instance. Average instance cost simply assigns each instance the average cost of all configurations that have completed runs on it. This approach requires a number of initial runs on each instance, so that there is data on which the initial clustering can be based. Of course, this clustering needs to be adjusted as new configurations are evaluated.

The second approach, using precalculated instance properties, does not update during the algorithm configuration process. We strive to maintain high interoperability, and keep algorithms and resources independent of one another, so that either of the two resources (or any other ones that might be implemented later) can be used with any clustering algorithm. We also use two different clustering algorithms: Complete Linkage Clustering (CLC) [2] and a hierarchical algorithm provided by R. CLC is a fairly simple, greedy clustering algorithm that starts by assigning each instance its own cluster and then proceeds to merge together the two clusters that are closest to each other. In this context, the distance of two clusters is the maximum of instance distances within the cluster, i.e. for clusters $x = (x_1...x_n)$ and $y = (y_1...y_n)$, $d(x, y) = max_{i,j}(d(x_i, y_j))$, with $d(x_i, y_i)$ being the euklidian distance of the instances x_i and y_j obtained via the resource base. The merging of clusters stops when the variance of a merged cluster rises above a predefined threshold.

Once this clustering is established, the racing method can now choose instances based on this categorisation, and can also adapt its evaluation and comparison of configurations to clusters instead of single instances. Our adapted version of ROAR chooses instances from clusters on a round robin principle, thus balancing the number of runs each configuration gets in each cluster. This ensures that every configuration is evaluated on a heterogeneous set of instances, and avoids the problem stated in the introduction.

On top of this, our instance prioritisation method tries to ensure that - in each cluster - instances that have proved themselves to be useful in distinguishing between configurations are prioritised over those that haven't. This means that when the racing method requests a new instance in a cluster, instances with a high variance in cost have a higher probability to be returned. A high variance in cost means that changes in the parameters of the algorithms have a high influence on performance. Low-variance instances, on the other hand, are less

[2] originated by Sorensen (1948).

useful for separating good from bad configurations, since most configurations' runs have similar cost on these instances. Runs on these instances do not provide much valuable information, but incur costs (i.e. wasted cpu time) anyway.

4 Experiments

To judge the effects of our additions to the ILS/ROAR configurator, we performed a series of experiments.

4.1 Setup

The target algorithm for all of our experiments is CPLEX [3], a solver for the mixed integer problem. It was optimised on a mix of the CLS, CORLAT, MASS, MIK and Regions200 instances used in [6] and [7] (a total of 410 instances). There were five experiments in total. Each consisted of 25 independent runs

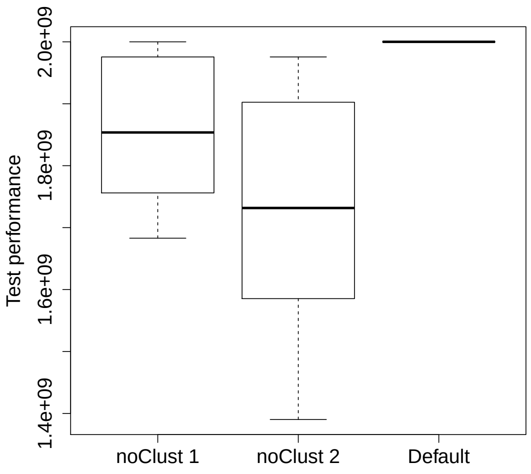

Fig. 1. Comparison of two identical experiments without instance analysis

of the configurator with a total tuning time of $25 \cdot 175140 = 4378500$ seconds, roughly 2 days for each run. A total of 64 CPUs was used, reducing the actual clock time for each run to less than an hour. The five experiments break down as follows: Two full experiments with ILS/ROAR without instance analysis, to serve as a baseline, and a simple way to judge variance in experiment results.

[3] The IBM ILOG CPLEX is a commercial mixed-integer-programming solver.

Then, two experiments with the CLC clustering algorithm, using average cost and precalculated instance properties as resources. Lastly, one experiment with R's hierarchical clustering algorithm and precalculated instance properties as a resource. All experiments were provided with the same initial "default" configuration as a starting point.

4.2 Results

The results of the first two experiments - without instance analysis - can be seen in figure 1. In both experiments, the configurator was able to find configurations that outrun the starting configuration "Default". It is, however, evident, that there is a great deal of random variation in these results, since two experiments with identical configurators produced relatively diverse results. This is not surprising, given that configurators are heuristic algorithms.

Figure 2 shows the remaining three experiments, where instance analysis was used. As before, all experiments were able to produce significant improvements over the default configuration. Based on this figure alone, it would appear that the most complex approach (hierarchical clustering with precalculated instance properties) slightly outperformed the very simple CLC approaches. Figure 3, however, shows that the differences in performance seen in 2 are well within the random variation inherent in the experimental setup, and its importance should therefore not be overstated.

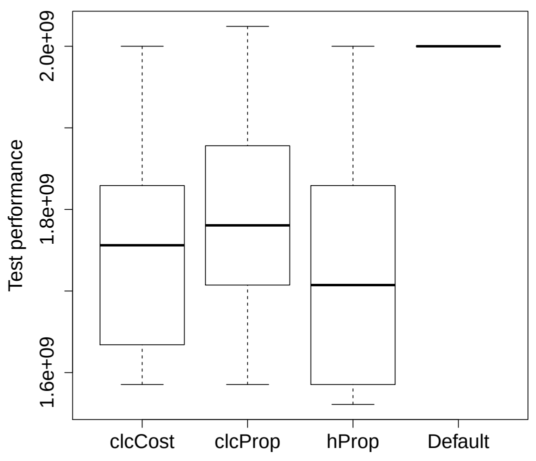

Fig. 2. Comparison of three different analysis setups. From left to right: CLC with mean instance cost, CLC with properties, and the hierarchical algorithm with properties

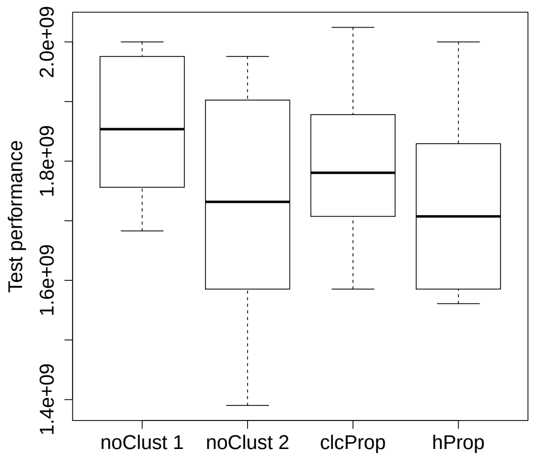

Fig. 3. Comparison of the worst and best experiments with and without instance analysis

5 Conclusion

The quality of results from an automatic configurator strongly depends on the set of training instances it works on. Therefore, an analysis of the instances, like the instance clustering described in this paper, avoids spending too much time on instances which do not provide new useful data. Our three approaches based on different cluster algorithms and different basis data all outran the default configuration as seen in 4.2 . In the direct comparison to the experiments without instance clustering the results are not obvious. Although instance clustering spares the configurator valuable configuration time worse results were achieved in one experiment. We therefore have to keep in mind that our used heuristic configurators are based on several random variables which can lead to a high variance in results.

Still, the analysis of the configurators instance set is a promising field of work. Besides our model-free approaches the idea of using model-based configurators for further work is a reasonable consideration.

References

1. Balint, Adrian; Diepold, Daniel; Gall, Daniel; Gerber, Simon; Kapler, Gregor; Retz, Robert: EDACC - an advanced platform for the experiment design, administration and analysis of empirical algorithms. 2011.
2. BW-GRiD: Member of the German D-Grid initiative, funded by the Ministry of Education and Research and the Ministry for Science, Research and Arts Baden-Württemberg.
3. Diepold, Daniel: Model-based Parallel Automated Algorithm Conguration. Ulm, Germany. 2012.
4. Hutter, Frank: Automated Conguration of Algorithms for Solving Hard Computational Problems. Vancouver, Kanada. 2009.
5. Hutter, Frank; Hoos, Holger H.; Leyton-Brown, Kevin: Sequential Model-Based Optimization for General Algorithm Conguration (extended version). 2011.
6. Hutter, Frank; Hoos, Holger, Leyton-Brown, Kevin: Automated conguration of mixed integer programming solvers. In Proc. of CPAIOR-10 (2010), pp. 186 - 202.
7. Hutter, Frank; Hoos, Holger, Leyton-Brown, Kevin: Parallel Algorithm Conguration. In Hamadi and Schoenauer [22], pp. 55 - 70.
8. Mugrauer, Frank: Automatisierte Parallele Algorithmenkonguration. Ulm, Germany. 2012.
9. Schulte, Juri: Analyse von Parallelen Automatischen Algorithmen-Konguratoren. Ulm, Germany. 2011.
10. Xu, Lin; Hutter, Frank; Hoos, Holger; Leyton-Brown, Kevin: Features for SAT. Vancouver, Canada. 2012.