CORA Cork Open Research Archive
Cartlann Taighde Oscailte Chorcaí

Title	Candidate selection and instance ordering for realtime algorithm configuration
Authors	Fitzgerald, Tadhg;O'Sullivan, Barry
Publication date	2019-03-14
Original Citation	Fitzgerald, T. and O'Sullivan, B. (2019) 'Candidate Selection and Instance Ordering for Realtime Algorithm Configuration', Fundamenta Informaticae, 166(2), pp. 141-166. doi: 10.3233/FI-2019-1798
Type of publication	Article (peer-reviewed)
Link to publisher's version	https://content.iospress.com/articles/fundamenta-informaticae/fi1798 - 10.3233/FI-2019-1798
Rights	© 2019 IOS Press All rights reserved.
Download date	2026-11-08 13:54:27
Item downloaded from	https://hdl.handle.net/10468/8115



UCC
University College Cork, Ireland
Coláiste na hOllscoile Corcaigh

Candidate Selection and Instance Ordering for Realtime Algorithm Configuration*

Tadhg Fitzgerald
Insight Centre for Data Analytics
Department of Computer Science
University College Cork
Cork, Ireland
tadhg.fitzgerald@insight-centre.org

Barry O'Sullivan
Insight Centre for Data Analytics
Department of Computer Science
University College Cork
Cork, Ireland
barry.osullivan@insight-centre.org

May 24, 2019

Abstract

Many modern combinatorial solvers have a variety of parameters through which a user can customise their behaviour. Algorithm configuration is the process of selecting good values for these parameters in order to improve performance. Time and again algorithm configuration has been shown to significantly improve the performance of many algorithms for solving challenging computational problems. Automated systems for tuning parameters regularly out-perform human experts, sometimes but orders of magnitude.

Online algorithm configurators, such as ReACTR, are able to tune a solver online without incurring costly offline training. As such ReACTR's main focus is on runtime minimisation while solving combinatorial problems. To do this ReACTR adopts a one-pass methodology where each instance in a stream of instances to be solved is considered only as it arrives. As such ReACTR's performance is sensitive to the order in which instances arrive. It is still not understood which instance orderings positively or negatively effect the performance of ReACTR. This paper investigates the effect of instance ordering and grouping by empirically evaluating different instance orderings based on difficulty and feature values.

*This paper is an extended version of work published in the Proceedings of the ACM Symposium on Applied Computing 2017

Though the end use is generally unable to control the order in which instances arrive it is important to understand which orderings impact ReACTR's performance and to what extent. This study also has practical benefit as such orderings can occur organically. For example as business grows the problems it may encounter, such as routing or scheduling, often grow in size and difficulty.

ReACTR's performance also depends strongly configuration selection procedure used. This component controls which configurations are selected to run in parallel from the internal configuration pool. This paper evaluates various ranking mechanisms and different ways of combining them to better understand how the candidate selection procedure affects realtime algorithm configuration. We show that certain selection procedures are superior to others and that the order which instances arrive in determines which selection procedure performs best.

We find that both instance order and grouping can significantly affect the overall solving time of the online automatic algorithm configurator ReACTR. One of the more surprising discoveries is that having groupings of similar instances can actually negatively impact on the overall performance of the configurator. In particular we show that orderings based on nearly any instance feature values can lead to significant reductions in total runtime over random instance orderings. In addition, certain candidate selection procedures are more suited to certain orderings than others and selecting the correct one can show a marked improvement in solving times.

1 Introduction

Modern search algorithms for solving computationally challenging problems are extremely complex. Due to this complexity, and the fact that no single configuration can perform best in all scenarios, algorithm creators will usually expose an often large number of parameters to the end user so that the behaviour of the solver can be customised at runtime. The choices available may include a choice of which heuristic to use, how many cuts to make in a mixed integer programming solver, or which restart strategy to use. Decisions made at this point can have a large impact on overall performance of the solver, sometimes by orders of magnitude [1, 2]. Furthermore, because the interactions between an algorithm's tens or possibly hundreds of parameters are often complex and counter-intuitive it is a daunting task to set all parameters correctly, even for a domain expert.

It is for this reason that the field of automated algorithm configuration has emerged [3, 4]. Automated algorithm configuration involves automatically selecting the most appropriate parameters for a certain set of instances such that the algorithm's performance is improved (running time, solution quality etc.). Using an algorithm configurator allows the end user to achieve strong performance gains without enduring the time consuming task of manually adjusting the parameters.

1.1 Offline Algorithm Configuration

There are two approaches to automated algorithm configuration, online and offline, each with their own pros and cons [5]. Currently the predominant method of algorithm configuration is the offline approach. Offline algorithm configuration can be broken down into two phases: the training phase and the testing phase. During the training phase a set of problem instances is used to learn a single configuration for the solver. This configuration is then used to solve all future unseen instances in the testing phase.

There are a number of state-of-the-art offline configurators that use various methods of configuration learning in the training phase. F-Race and Iterated F-Race adopt a racing approach to algorithm configuration [6, 7]. F-Race races configurations from a predetermined pool against each other. This pool is created either by a full factorial design or random sampling of the configuration space. Each race or iteration is run on a problem instance. After each iteration, configurations that are shown to be statistically weaker than the current incumbent, using a non-parametric Friedman test, are removed. This process continues until only a single configuration remains or the allotted configuration time is exhausted. Iterated F-Race improves upon F-Race by removing the predetermined pool and instead sampling configurations from promising areas of the configuration space after each iteration. Previous iteration results are used to bias the sampling in the current run towards these promising areas.

ParamILS uses stochastic local search to explore the configuration space around a given starting configuration, e.g. a default configuration [4]. In order to avoid local optima a perturbation phase is applied after the local search phase. ParamILS also diversifies by 'jumping' to a different starting configuration with a certain probability.

Gender-based genetic solvers form the core of the GGA algorithm configurator [8]. Similar to regular genetic approaches, configurations are combined using crossover while mutation is used to diversify the configurations discovered. GGA also uses two 'genders', competitive and non-competitive, when performing crossover (mating). Only the best, top N%, of the competitive group are able to mate in the next iteration. This puts a selection pressure on the best configurations. Configurations that have been in the pool for more than a specified number of iterations are removed from the pool.

SMAC uses a Sequential Model Based Optimisation (SMBO) approach to algorithm configuration [1]. The configurator builds a random forest model which uses instance features to predict performance. The configurations with the best expected performance improvement based on this model are then selected for evaluation against the current incumbent.

Instance Specific Algorithm Configuration (ISAC) partitions instances into clusters based on their feature vectors[9]. The instance clusters are formed using the g-means clustering algorithm (an extension of k-means which automatically selects the k value). The previously outlined configurator GGA is used to learn a configuration for each individual cluster. When a new instance arrives the euclidean distance between it and the centres of the clusters is computed and

the previously learned configuration from the nearest cluster is used.

Offline configuration techniques have been shown to work time and again, occasionally improving performance by multiple orders of magnitude [1]. There are a number of advantages to performing configuration offline. As the configurator is allocated a fixed amount of configuration time, there is more available overhead which allows instances to be solved multiple times with multiple configurations. This allows for a better understanding of the corresponding distribution and takes variance into account. The ability to revisit instances means fewer training instances are required. The larger configuration budget also allows for more expensive machine learning techniques to be used.

1.2 Online Algorithm Configuration

While offline configuration has had its share of successes, there are a number of drawbacks to using offline configuration techniques. Firstly, a representative sample of instances must be available. For previously unseen problems these instances might not be readily available. If there is an insufficient number of training instances, or the training instances are not representative of those that will be encountered during testing, then it is possible for offline techniques to overfit the training instances. As the configuration ceases to be updated after the initial training phase, any changes in the instance-types encountered will not be considered unless retraining occurs. Secondly, an expensive training period is required, after which the configuration stops improving regardless of the number of instances encountered. It is possible for the time allocated to configuring the solver to eclipse any improvements in solving time.

With these limitations in mind we turn our attention to online algorithm configuration. It has been shown that the downtime between solving instances could be used to find continually improving search heuristics for constraint programming without using an explicit training period [10]. Our recent work in the area of online algorithm configuration [11, 5] has shown that configurations can be improved without downtime while avoiding some of the issues faced by offline configurators outlined above. Our online configurator ReACTR requires no previous instances or offline training period. Instances are solved only once so the configuration incurs little overhead compared to offline configurators which repeatedly solve the same instances during training. Configurations are constantly evolving so if the type of instances change the configurator is able to adapt and find more suitable configurations. In this way ReACTR is able to reduce the total runtime required to solve problem instances to optimality while also improving the configurations used.

Online algorithm configuration is closely related to the online algorithm selection problem[12, 13]. Here an algorithm selection system is trained on the fly as it processes a stream of instances.

1.3 Objectives and Structure of this Paper

The goals of this paper are twofold: first, to explore the effect of instance ordering and grouping on the ReACTR configuration system, and second, to evaluate various candidate selection policies within ReACTR. The interaction between both of these is also investigated; do some instance orderings achieve better performance when using certain candidate selection procedures?

ReACTR is constantly updating and evolving throughout the lifetime of the configuration process. While this is certainly an advantage, it may also cause problems. ReACTR visits each instance only once and updates its leaderboard between instances. The configurator is therefore extremely sensitive to the order in which instances arrive to be processed. One of the two primary aims of this paper is to investigate and try to understand how exactly, and to what extent the ordering and grouping of instances affects ReACTR.

It is possible to imagine a run of adversarial instances designed to hinder the performance of the ReACTR system. One way this might be achieved would be to order instances such that the best configuration is changing rapidly. ReACTR relies solely on historical performance and does not use instance features when selecting which configuration to run. As such this type of ordering could be used to disrupt ReACTR's ranking system and selection procedure. If all configurations are ranked similarly selecting amongst them may prove difficult for the system. This flaw could be mitigated by selecting some of the most recent winners in addition to configuration ranking. The advantage of this is that it detects changes in the incumbent more rapidly than ranking systems alone. We show that in certain cases this approach is very effective. Another option would be to use an instance specific candidate selection procedure similar to ISAC, though integrating this to the ReACTR system would come at the cost of additional overhead. Another pitfall which ReACTR is susceptible to is overfitting. This happens when a long stream of instances are all solved by a single configuration or a small group of configurations. More generally applicable configurations (such as the default configuration) may be removed by ReACTR as they are not winning any races. If the type of instances encountered shifts suddenly the system will recover more slowly as it is not able to generalise as well. One solution to this may be to keep previous winners or incumbents and reintroduce them periodically (or when concept drift is detected), regardless of rank.

Conversely, it is possible that certain orderings of the instances may actually improve configurator performance by providing more information at crucial points in training. The concept of varying instance ordering has been explored in machine learning to increase learning rate [14] and in algorithm configuration to improve scaling performance [15]. Intuitively, grouping similar instances together should result in improved performance as similar instances tend to favour the same types of configurations. There are multiple ways to cluster similar instances; sorting by feature values (as we have done in this paper), using a homogeneity measure [16] or by adopting a clustering approach[9, 17]. This paper aims to explore what effect this instance ordering has on the ReACTR

system and how the system can take advantage of these insights.

ReACTR's parallel racing mechanism provides it with a way to evaluate multiple configurations side by side. This is what allows ReACTR to function in an online fashion. An important part of this parallel racing mechanism is being able to correctly identify which configurations should be evaluated on each instance. Parallel algorithm portfolios face a similar challenge when selecting which solvers to run on each instance. Portfolios generally aim to maximise coverage of the number of instances by running complimentary solvers [18]. This is often done by looking at instance features and predicting which solvers will work well on which instances[19]. ReACTR does not use instance features, but instead relies on past performance data and a ranking system to determine which candidates to select for each instance. The candidate selection procedure plays a large role in the ReACTR system and its overall performance. We explore various combinations of performance metrics to decide which provides the best performance.

The paper is organised as follows. Section 2 outlines the core components of the ReACT and ReACTR systems. Section 3 outlines the experimental setup and the datasets. It also explains in detail the candidate selection policies and instances orderings used. Section 4 focuses on the analysis of the effect of ordering, grouping and candidate selection on a dataset consisting of precomputed runtime results for a fixed set of solvers. Section 5 presents the results of experiments conducted using full ReACTR runs to determine the effect of instance ordering and grouping with a dynamic configuration pool. Finally, Section 6 outlines our conclusions and potential extensions of this work.

2 The ReACTR System

2.1 ReACT

We introduced the concept of realtime algorithm configuration using parallel racing in our earlier ReACT work [11]. The goal of ReACT is to solve a stream of instances as quickly as possible while also improving the solver configuration. ReACT aims to incur very little overhead. Once an instance has been solved, there is no incentive to solve it again, for this reason instances are visited once and only once during a configuration run.

ReACT races multiple configurations in parallel on the same instance using the multi-processor cores commonly available in modern computers. After each race, the winning configuration is deemed to be the configuration that solved the instance the quickest. All other runs are stopped once the instance has been solved so as not to incur any additional overhead. A pool of n configurations is maintained, where n is typically the number of cores in use. Wins and losses between pairs of configurations are tracked in a matrix. Underperforming configurations are removed by means of a simple statistical test. Configurations are removed from the pool when a configuration has run a minimum number of races against another and is being dominated by the other configuration, e.g.

the dominant configuration has double the number of wins of the weaker configuration. The removed configuration is replaced by another sampled randomly from the configuration space. Though the techniques used in ReACT are simple, it shows that this realtime configuration methodology works and laid the ground work for future improvements.

The more recent ReACTR system improves on all aspects of the ReACT configurator [5]. ReACTR can be broken into three main parts: the configuration leaderboard, the candidate selection policy, and the configuration replacement procedure.

2.2 Configuration Leaderboard

ReACTR replaces ReACT's small configuration pool and simple statistical test with a leaderboard able to track more configurations and rank them more robustly. This leaderboard is backed by the TrueSkill ranking system. TrueSkill (TS) is a Bayesian ranking system developed by Microsoft Research [20]. It was originally developed to provide close matches for XBox video game players. In gaming, TrueSkill is used to match the skill level of players who may not have competed against one another in order to create balanced games. In ReACTR, it must quickly assess the relative performance of solver configurations while they are constantly added and removed from the configuration leaderboard. TrueSkill measures both a player's average skill, μ, and its certainty in the assigned skill rating, σ. TrueSkill uses a Gaussian belief distribution to model a player's skill.

Updates to a player's μ and σ are performed after each competition. The magnitude of the update depends on the skill difference between players and the confidence the system has in the assigned ratings. For example, if a strong player defeats a weak player the result is unsurprising and so there is a minimal adjustment to μ (assuming similar σ values). However, if a weak player beats a strong player this will cause a large shift in the μ score for both players. The system becomes more confident in its assigned scores with the more races that are run and, as such, the σ value only decreases. These updates are extremely fast and so the overhead in updating ReACTR's leaderboard is negligible from a performance perspective.

The systems leaderboard is typically initialised with configurations sampled uniformly from the space of feasible configurations. The default configuration of the solver being configured is also included in this leaderboard and given a slightly higher TrueSkill ranking than other configurations. Aside from this, the system is typically not warm started, though doing so is possible by including desired configurations in the initial leaderboard. In order to limit the memory impact of storing configurations we limit the leaderboard size to thirty configurations.

2.3 Candidate Selection Policy

ReACTR's candidate selection policy is responsible for choosing which configurations to run on an incoming instance. Given that only a limited number of configurations may be run in parallel, the challenge lies in balancing the exploration of newly added, untested configurations from the leaderboard while maintaining good performance by using 'proven' configurations. This is known as the exploration versus exploitation tradeoff and is a well-studied issue [21, 22]. Drawing on previous research, the current version of ReACTR adopts an Epsilon-Greedy approach for selecting candidates from the internal leaderboard. Here one third of the chosen configurations are 'good' while the other two thirds are randomly selected. Here 'good' is defined as the configurations with the highest score using the TrueSkill performance metric.

2.4 Removing and Replacing Configurations

In order to explore the space of possible configurations it is necessary to remove underperforming configurations. ReACTR uses TrueSkill's skill and confidence metrics to remove weak configurations without risking the removal of good configurations. A configuration is removed when it is at the bottom of the leaderboard (it's TrueSkill score falls below the median) and TrueSkill's confidence reaches a predefined threshold.

Configurations that have been removed must be replaced. ReACTR continues to use the random sampling technique used in ReACT which provides good diversification in the configuration leaderboard. This is supplemented by an intensification procedure based on genetic algorithms. Top configurations, based on TrueSkill ranking, are combined by uniform crossover to create new configurations with parameters taken from both parents. A small mutation chance of mutation(5%) is also used to allow a parameter take on a value which neither parent has. This allows search to move towards the more promising areas of the configuration space. ReACTR generates an equal number of configurations using randomisation and crossover. These procedures favour discrete parameters but are able to naively handle continuous parameters also by discretising the space at runtime.

3 Experimental Method

3.1 Candidate Selection

One of our aims is to more fully understand the candidate selection procedure used in ReACTR. ReACTR's current candidate selection policy uses TrueSkill as its performance metric. In our experiments we evaluate a number of different performance metrics, and various methods of combining them. The Epsilon-Greedy approach is still used, so a proportion of candidates selected are always random to ensure a certain level of exploration within the parameter pool. In

addition to these, a number of 'good' candidates are selected based on various performance metrics as discussed below.

In addition to testing TrueSkill as a performance metric we also look at a number of simpler metrics based on the number of wins (win count), and how recent those wins were measured. In this context a winner is defined as the configuration that outperformed its competitors by finishing more quickly. Last Winners (LW) keeps a list of the most recent winners. It then selects the n most recent unique configurations. Win Count (WIN#) is the number of times a certain configuration has performed best. It is important to note this is not normalised based on the number of runs the configuration has taken part in. Win Percentage (WIN%) is similar to Win Count but normalises for the number of runs in which a configuration has participated. This normalisation makes Win Percentage a better metric as it is no longer biased towards configurations with more runs.

Another complimentary performance metric is evaluated, Defeats (DEF). Defeats must be used in conjunction with a primary performance metric. Defeats chooses the n configurations that have beaten the incumbent selected by the primary metric the most.

These selection procedures are applied in the order they are listed. For example 2TS2LW2RAND will select two configurations using TrueSkill ranking(TS), two using the Last Winners procedure (LW) and finally two at random(RAND). If a configuration has already been selected by a previous procedure then the next unselected configuration is chosen. For example Last Winners will select the next most recent winner that is available.

A number of baseline metrics are also provided. Due to the differing nature of the runs, different baselines are proposed for static and dynamic datasets. In the static case, three metrics are provided. First, the *Oracle* is the virtual best system, which selects the best solver for every instance. This is the best performance that is possible to achieve. *Single Best* shows the best single solver that minimises the overall total solving time. *Random* selects six solvers (we're assuming six CPU cores are available) at random for each instance and chooses the best solving time from these. In the dynamic case the baseline is the average running time of the untuned solver.

3.2 Datasets and Instance Generation

For these experiments we use three different datasets. Two datasets, PROTEUS-2014 and SAT12-ALL are taken from the Algorithm Selection Library (ASLib) [18, 23, 24]. These are static datasets where run times for all solvers are precomputed and the candidate selection procedure is performed as a processing step without doing a full ReACTR run. In this case we treat each individual solver as a different "configuration" and candidate selection is performed on these. The pool of "configurations" is fixed as the removal and replacement procedure normally seen in ReACTR are not employed. Excluding ReACTR's removal and replacement procedure also allows us to study the effect ordering and candidate selection in isolation. The caveat to this is that the interaction between

these procedures cannot be studied and as such full ReACTR runs may behave differently. However, it is necessary to use static datasets as running the full ReACTR system requires a large number of CPU hours and does not readily parallelise. Though the results are precomputed, we process the stream in the same way as ReACTR one instance at at a time albeit without the removal and replacement procedures. This processing step is quick, therefore the static results are based on 100 runs which allow for far more statistically significant conclusions. Using static datasets to help focus on interesting results is a necessity as each ReACTR run can take a number of hours or even days. A single dynamic dataset, Combinatorial Auctions[1], is also evaluated by a full ReACTR run.

The first static dataset PROTEUS-2014 comprises 4021 constraint satisfaction problems (CSP) solved using 22 different solvers. We use Mistral as the default solver. This is in keeping with the methodology used in the paper which created this dataset [18]. Instances that the default solver solved in less than two seconds were filtered out. Additionally any instances where all solvers hit the 3600 second cut-off time were removed. This left 2595 instances for analysis which were neither too easy nor difficult. Four different orderings of these 2595 instances were then considered: lexicographic, shuffled, easy-to-hard and hard-to-easy. Lexicographic orders the instances based on the lexicographical ordering of the instance file paths. This is important because the way the dataset was formed means similar instances tend to be clustered together in the same folders e.g. 8-Queens would be next to 9-Queens in the n-Queens folder. Shuffled completely shuffles all instances so they are randomly ordered. Easy-to-hard and hard-to-easy sort the instances from fastest solving time to slowest, and vice-versa, using the default solver, Mistral. For our feature ordering experiments we used a subset of the PROTEUS-2014 dataset comprising 623 instances. This subset was chosen so that all feature values were present and did not need to be computed. All 198 instance features given as part of the ASLib dataset are used; these are described in more detail in Section 3.3. These feature values are used to sort the instances in both ascending and descending order.

The second static dataset, SAT12-ALL, contains a mix of 1614 Boolean Satisfiability (SAT) instances taken from SAT competitions. These are solved using 31 different solvers. Lingeling is selected as the default solver for this dataset. Lingeling is a highly parameterised SAT solver which has performed well in recent configurable SAT solver challenges[2]. Again, any instances that take less than 2 seconds to solve with the default solver, Lingeling [25], or are unsolved by any solver within the 1200 second cutoff time, were removed. The remaining 1474 instances were again ordered in the four orderings outlined above. Similar to PROTEUS-2014, for our feature ordering experiments we only used a subset of the instances where all instance feature values are available. This gave a dataset comprising 721 instances. We ordered by all 115 instance features, which will be described in Section 3.3, in both ascending and descending order.

[1] `http://www.cs.ubc.ca/~kevinlb/CATS/`

In addition to the two static datasets, a single dynamic benchmark based on combinatorial auctions was also used. This dynamic benchmark does not have pre-calculated run times and must be solved by the ReACTR system to determine the runtimes. Due to computation time involved in running ReACTR over a large number of instances, each run using the dynamic dataset is evaluated only 10 times. All dynamic experiments were run on a system with 2×Intel Xeon E5430 processors(2.66Ghz) and 12 GB RAM. The system has 8 available cores but only 6 are used so as to allow room for background processes and other overhead without affecting timing. The Algorithm Configuration Library(ACLib) [26] framework was used to run the dynamic experiments.

Four different combinatorial auction domains are combined to create the dynamic benchmark based on combinatorial auctions. These instances are generated using the Combinatorial Auctions Test Suite (CATS) [27]. The four domains generated are arbitrary, paths, regions and scheduling. Arbitrary and regions instances were both generated using 100-100 goods and 100-2000 bids, paths instances have between 512 and 2048 goods with 3000-10000 bids, while scheduling instances have 128-256 goods and 3000-10000 bids. These parameters were chosen in order to create instances that proved challenging for the mixed integer programming (MIP) solver used, CPLEX [28]. CPLEX 12.6 is used with 74 discretised parameters as provided by ACLib. The first 200 instances for each domain, that were solvable within 30 to 600 seconds using the default CPLEX configuration, are then merged to create a benchmark with 800 challenging but solvable instances. When solving using ReACTR a cut off time of 180 seconds was used, which allows room for the configurator to improve over the default solver configuration.

3.3 Features

The SAT12-All dataset uses 115 SAT instance features that are used by SatZilla in the 2009 SAT competition [29]. For the sake of brevity we will not exhaustively list all features, however, they can be divided into subgroups: problem size features, variable-clause graph features, variable graph features, balance features, proximity to Horn formula, DPLL probing features, local search probing features, survey propagation features and clause learning features. A technical report describing these features in more detail is available [30].

The PROTEUS-2014 dataset contains instances and features used in the Proteus hierarchical portfolio of solvers [31]. Proteus uses both SAT and CSP solvers, sometimes encoding a CSP problem in SAT and solving using a SAT solver if this is expected to improve performance. Therefore, the PROTEUS-2014 dataset contains a mixture of both CSP and SAT features. There are 36 CSP features which were originally used in CPHydra [32]. These include statistics about the domain sizes, the type of constraints and the progress of the Mistral solver when run for 2 seconds. The SAT features used for the PROTEUS-2014 dataset are similar to those of SAT12-All outlined above. The one notable difference is that the features are calculated from multiple different SAT encodings of the CSP instances (support, direct order and direct).

4 Experiments on Fixed-set Solver Datasets

4.1 Lexicographical and Runtime-based Ordering

This paper has two goals, to explore which candidate selection strategies work best, and also what effect the instance ordering has on the overall solving time. To do this different performance metrics (described in Section 3.1) are used and combined in a number of ways and then evaluated on multiple instance orderings. Figure 1 shows box plots representing the solving time using various selection mechanisms on the SAT12-ALL dataset which has been ordered lexicographically. The sampling methods are ordered by their median values, which is indicated in the boxplot by the red central line. The blue box shows the first and third quartiles (the 25th and 75th percentiles). The lower whisker shows $Q1 - 1.5 \times IQR$, inter-quartile range (IQR), while the upper whisker shows $Q3 + 1.5 \times IQR$, where $Q1$ and $Q3$ are the first quartile and third quartile, respectively, and the IQR is the difference between them. Outliers are marked with black crosses.

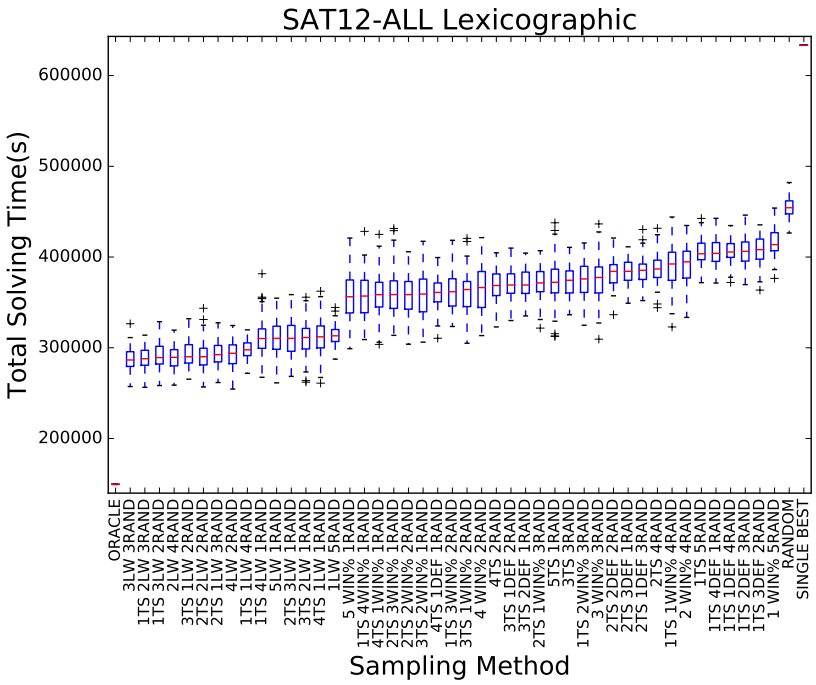


Figure 1: Comparison of all candidate selection methods on the SAT12-ALL dataset ordered lexicographically (Avg. of 100 runs).

Three baselines are included to give some context to the results. *Oracle* chooses the best possible solver for each instance. *Random* selects six solvers at random for each instance and logs the best time achieved from the six.

SingleBest is the solver which has the lowest overall solving time over all instances. All of the selection methods shown outperform both the *Random* and *SingleBest* baselines which shows that using any of the ReACTR selection methods alone even without the ability to modify or introduce new solvers or configurations still gives an improvement and is a worthwhile endeavour. Though not shown, this result holds across all orderings of both PROTEUS-2014 and SAT12-ALL datasets.

Figure 1 shows a jump in solving time when transitioning from 1LW 5RAND to 5WIN% 1RAND. Prior to this point all selection policies used Last Winners (LW) as a selection component, after the point combinations of TrueSkill, Win Percentage and Defeats are used instead. It is clear that having Last Winners as part of the selection policy is important for SAT instances that are lexicographically ordered. When the instances are ordered lexicographically rapid changes in the domain type occur, for example when switching from a folder containing hardware verification to cryptography instances. By using the last winner these changes are detected quickly rather than waiting for a TrueSkill score or Win Percentage to rise sufficiently for the candidate to be selected by those metrics. Supporting this hypothesis is that none of the other SAT orderings, which are shuffled instance-wise, show this sharp jump.

There is a smaller jump also visible between the ninth and tenth box plots (1TS 1LW 4RAND and 5LW 1RAND) which appears to be caused by the reduction in the number of random candidates. Last Winners, in the SAT lexicographical case, seems to need a larger number of random candidates included. Since the domains encountered are changing rapidly, a heavy emphasis on exploration within the leaderboard is required. Random selection provides this exploration and allows the selection policy to quickly discover the best solver for the current instance type, while using Last Winners allows the best solver to be kept and used for upcoming instances. This behaviour is not typical for the other orderings or datasets where normally more exploitation using only one or two Random exploration candidates is favoured.

Note that selection policies that include Defeats appear to under-perform. When examined more closely Defeats adds little improvement over selecting randomly. In general, it appears the majority of any good performance observed when Defeats is used can be attributed to the TrueSkill part of its composition.

Figures 2 and 3 show the best selection policies from each individual ordering evaluated on all of the other instance orderings for both PROTEUS-2014 and SAT12-ALL respectively. The y-axis is shown in a log scale in order to include the baseline results. Again we see that all selection policies outperform the Random and Single Best baselines. In both cases we see that instances that are ordered lexicographically are solved faster than all other orderings using the optimal selection policy. Shuffled instances, though not solved fastest, have the smallest deviation in solving time. This is most obvious in the case of Proteus Shuffled.

The lexicographically sorted Proteus dataset is solved fastest using a combination of TrueSkill and Last Winners. This is in keeping with what was shown in Figure 1 and for possibly the same reasons outlined previously. The

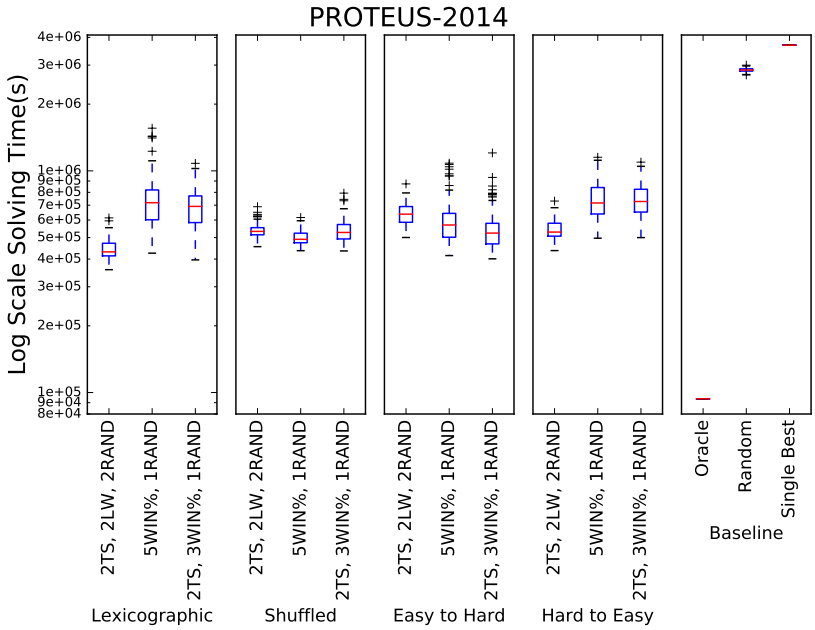


Figure 2: The solving time for the best performing candidate selection mechanisms for each PROTEUS-2014 ordering when evaluated on all other orderings (Avg. over 100 executions).

other orderings appear to express a preference for certain candidate selection models as well regardless of dataset. Shuffled instances from both Proteus and SAT are are solved most quickly using a Win Percentage-based system (5WIN% 1RAND and 1TS 3WIN% 2RAND, respectively). Similarly, selection policies using TrueSkill and Last Winners perform best on instances that are ordered from hard-to-easy (2TS 2LW 2RAND for Proteus and 3TS 1LW 2RAND for SAT). The only ordering which bucks this trend is easy-to-hard, favouring a TrueSkill and Win Percentage-based model for Proteus (2TS 3WIN% 1RAND) but a TrueSkill and Last Winners approach for SAT (3TS 1LW 2RAND).

Some commonality amongst the most preferred selection policies is visible also. Proteus Lexicographical and Hard-to-Easy fare best using two TrueSkill, two Last Winners and two Random, while both SAT Easy-to-Hard and SAT Hard-to-Easy agree on three TrueSkill, one LastWinner and two Random as the optimal selection policy. This may signify that these ratios are the right balance for those particular datasets. Unfortunately there is no general consensus on ratios across datasets, though Proteus Hard-to-Easy does have 3TS 1LW 2RAND as a second choice suggesting some commonality is present.

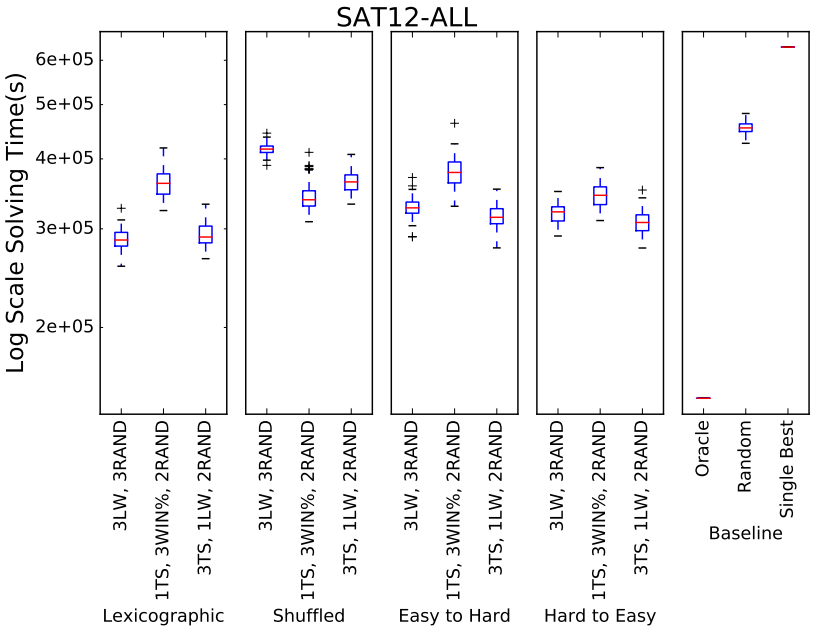


Figure 3: The solving time for the best performing candidate selection mechanisms for each SAT12-ALL ordering when evaluated on all other orderings (Avg. over 100 executions).

4.2 Feature-Based Ordering

Our initial ordering experiments showed that instance ordering does make a difference and positive improvements in the total solving time can be achieved based on ordering alone. Here we extend these ordering experiments to investigate the effect of ordering instances based on their feature values. Instance features can reveal much about the instances being processed such as their size, structure, and how the instance changes when a solver is run on it for a very short period of time; we refer to running a solver in this way as 'probing'. Most features can be computed extremely quickly, often by just parsing the instance file. Understanding how ReACTR performs on orderings based on different instance features provides a greater fundamental understanding of the configurator. These experiments show where ReACTR achieves its best and worst performance in certain domains.

Getting these insights leads to a greater understanding of how ReACTR will perform in real world scenarios. Companies are often faced with a stream of increasingly large problems to solve. For example a ride-sharing company might see the stream of instances it must solve grow in size and complexity as the company expands over time, or there can be similar effects between off-peak and rush-hour periods at a daily level. Similarly a factory may face

increasingly difficult scheduling problems as the company employs more staff or takes on more orders. Though there is no direct control over the type of instances encountered it is still important to be aware of how the configurator responds to the size of certain features increasing and decreasing so as to avoid any pitfalls.

The previous experiments have focused on which of the selection policies is most preferred whereas the goal of these experiments is to study the effect of instance feature ordering. As such we use a single candidate selection policy for the experiment: 2TS 2LW 2RAND comprising two TrueSkill, two Last Winners and two Random selectors. This candidate selection policy was chosen as it performed best in two of the previous PROTEUS-2014 experiments. Candidate selection policies containing a mixture of TrueSkill and Last Winner also performed well on the SAT12-All dataset making this a good compromise choice for both datasets being investigated.

The instances in the SAT12-All dataset has 115 unique instance features. These are ordered both ascending and descending to give a total of 230 different instance orderings. Each instance ordering is evaluated 100 times. We compared the distribution of total runtimes for each instance ordering against the distribution given by 100 runs of both Random and Easy-to-Hard ordering using a statistical hypothesis test (Student's t-test). Of the 230 different SAT feature orderings 223 were statistically better (P=0.01) than random ordering while 200 were statistically better than the Easy-to-Hard ordering. No ordering was statistically worse than Random, though four orderings did have worse average runtimes (UNARY and POSNEG_RATIO_CLAUSE_max sorted both ascending and descending). Eight orderings were statistically worse than the Easy-to-Hard ordering (POSNEG_RATIO_CLAUSE_min, POSNEG_RATIO_VAR_min (ascending and descending), gsat_FirstLocalMinStep_CoeffVariance, POSNEG_RATIO_CLAUSE_max (ascending and descending) and UNARY (ascending and descending)).

Figure 4 shows the average cumulative solving time for the instance feature ordering, with the best and worst performance as well as the baselines Random ordering, Easy-to-Hard ordering and Hard-to-Easy ordering. Even the worst feature ordering was only 1.9% slower than Random while the best feature ordering was 28.9% faster than Random. This implies that the potential gains of ordering instances by any feature value greatly outweighs the potential penalty associated with ordering on one of the few features that perform worse than Random. In other words, ordering instances randomly seems to be the worst thing one can do: it is important to order instances in a way that recognises instance size or structure.

Figure 5 shows box plots for the performance of the best ten orderings, the worst ten orderings, and three baselines (Random, Easy-to-Hard and Hard-to-Easy). Features that are appended with DESC are sorted in descending order. It is immediately obvious that there is a large gap between the best performing orderings and the worst. This is to be expected given the low p-values seen when statistically comparing these distributions against Random. What is somewhat surprising is that the disparity between the performance of the baselines and the worst performing feature orderings is quite small. While

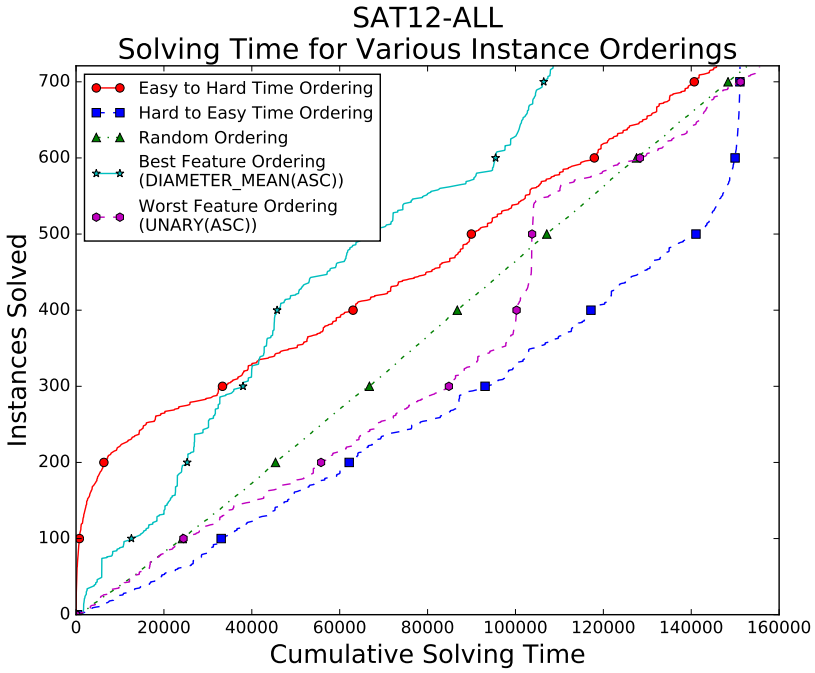


Figure 4: SAT12-All: Cumulative runtime graph for best and worst feature orderings with baselines (Avg. over 100 executions).

Easy-to-Hard outperforms Random, and Random outperforms Hard-to-Easy – which agrees with what was shown in previous experiments – all of the baselines are scarcely better than even the worst-performing ordering based on features.

Looking more closely at the features themselves we can see that statistics about the number of learned clauses (sorted descending) dominate the top ten. These clause learning features are based on a two second of run Zchaff_rand. DIAMETER_mean is the mean diameter of the variable graph. VCG_CLAUSE_max is a Variable-Clause Graph feature for the maximum clause node degree. cluster_coeff_mean is the mean clustering coefficient of the Clause Graph. lobjois_mean_depth_over_vars is a DPLL Probing feature which gives an estimate of the search tree size. This suggests that focusing on the constrainedness of instances, and specifically considering more tightly constraint instances first, is a good strategy. Instances of this kind provide greater opportunity to learn, since the relative strength of different solver configurations will be more clearly discernable. If one focuses on instances that are easy for all configurations, then there is little to distinguish them.

Visually inspecting the orderings produced by sorting on these features we see that similar instances tend to cluster together (not shown). This is akin to what we saw in the lexicographical ordering of our previous experiments.

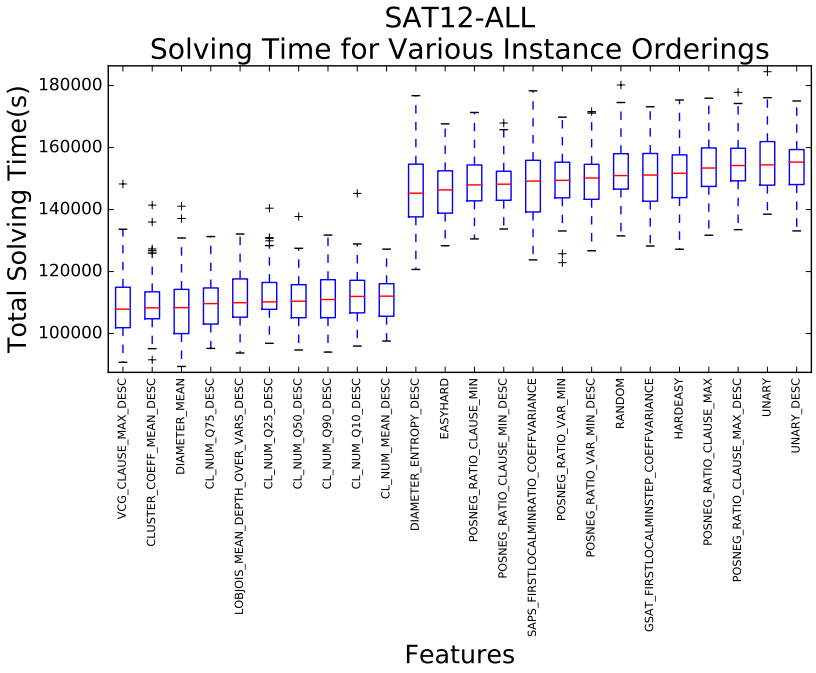


Figure 5: SAT12-All: Box-plots of the ten best, ten worst and three baselines (Avg. over 100 executions).

However, feature based ordering is more powerful than lexicographic ordering in that it does not rely on instances being within the same folder to group similar instances e.g. crypto instances from the SAT 2007 competition will still appear near crypto instances from the SAT 2009 competition despite not being in the same directory.

We see that the sorting order (ascending or descending) is important for the top performing features. For example 'DIAMETER_mean' sorted ascending is the best performing feature taking 109k seconds, however 'DIAMETER_mean' sorted descending is ranked 145th (132k seconds). It should be noted that both are still statistically better than all baselines. The opposite seems to be true of the poorly performing features whose runtime is close to that of the Random ordering. Both ascending and descending orderings of 'UNARY', 'POSNEG_RATIO_CLAUSE_MAX' and 'POSNEG_RATIO_VAR_MIN' are in the worst ten performing.

Turning our attention to the PROTEUS-2014 dataset again we see that a large percentage of feature orderings outperform the random ordering. Of the potential 396 orderings (198 features sorted both ascending and descending) 376 have a lower average total solving time. After analysing these results using Student's t-test we find that 310 feature orderings give a statistically significant

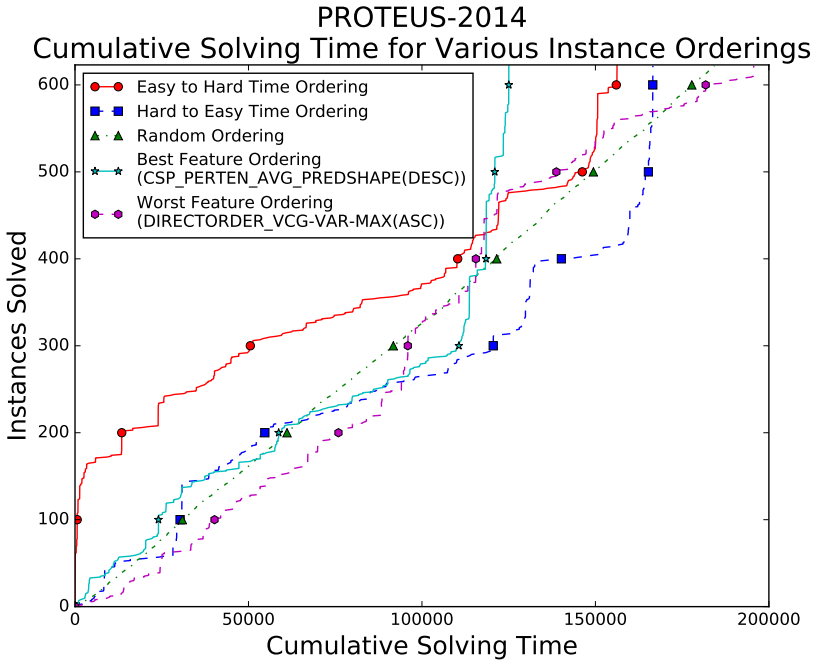


Figure 6: PROTEUS-2014: Cumulative runtime graph for best and worst feature orderings with baselines (Avg. over 100 executions).

improvement over the random ordering while only four where statistically worse (direct_cluster-coeff-mean_DESC, support_cluster-coeff-mean_DESC, csp_dyn_log_stdev_weight, directorder_VCG-VAR-max). The ordering with the fastest total solving time was 32.1% faster than random ordering while the worst was 6.2% slower. When comparing against Easy-to-Hard, we find that 74 feature orderings perform statistically better while 221 are statistically worse. Figure 6 shows the average cumulative solving time for the best and worst feature ordering in addition to the three baselines. The progression for best ordering is interesting in that it follows the gradient of the random ordering for the first 300 instances before solving the remaining 323 instances extremely quickly. This somewhat contradicts the idea that solving the easiest instances first is always desirable. However, it is important to remember that these are static experiments with a fixed pool of solvers. In the dynamic case it may be desirable to solve the easiest instances first and by doing so learn more configurations quickly.

The box-plots in Figure 7 show that unlike SAT12-All the ten worst orderings do perform worse than the baselines though not too much worse than Random ordering. The features for the PROTEUS dataset are interesting in that they contain a mixture of CSP and SAT features. The SAT features are replicated for three different encodings used when encoding the CSP instances to SAT (DIRECT, DIRECTORDER and SUPPORT). These encodings are prepended

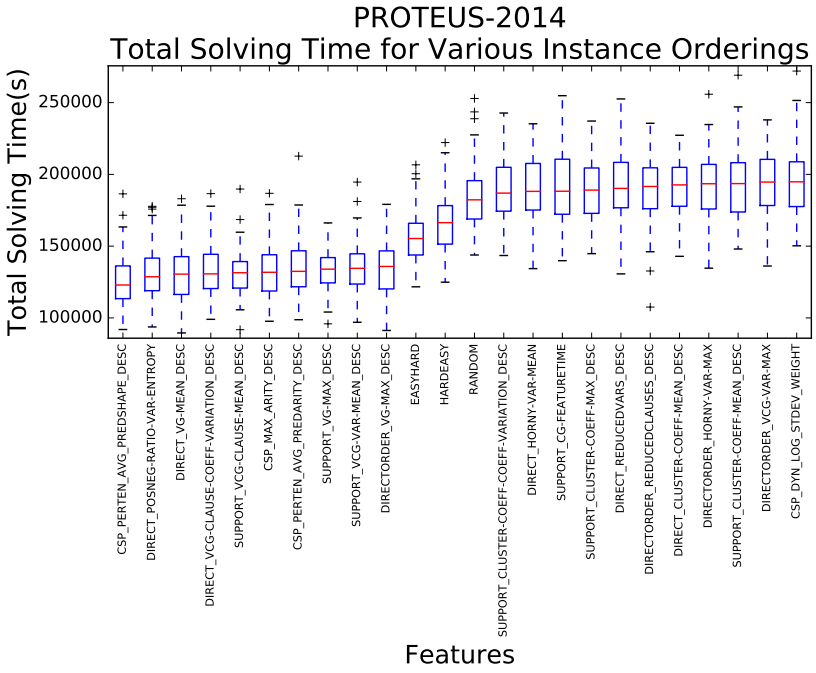


Figure 7: PROTEUS-2014: Box-plots of the ten best, ten worst and three baselines (Avg. over 100 executions).

to the feature names to indicate the encoding used, while CSP features have 'CSP' prepended to them. Both the ten best and ten worst contain at least one of each type of feature. It is also interesting to note that the PROTEUS and SAT datasets do not agree on the best features to order by, in fact CLUSTER_COEFF_MEAN_DESC is the second best feature ordering in SAT whereas it is the third, fourth and sixteenth worst in PROTEUS for SUPPORT, DIRECT and DIRECTORDER encodings respectively.

Variable-graph node degree statistics seem to be important for ordering the PROTEUS dataset. The maximum node degree occurs twice in the ten best orderings (DIRECTORDER_VG-MAX_DESC, SUPPORT_VG-MAX_DESC) while the node degree mean also appears in the top ten (DIRECT_VG-MEAN_DESC). Statistics relating to the Variable-Clause graph occur three times in the top ten (SUPPORT_VCG-CLAUSE-MEAN_DESC, DIRECT_VCG-CLAUSE-COEFF-VARIATION_DESC, SUPPORT_VCG-VAR-MEAN_DESC). These are both interesting since the variable graph and the variable-clause both relate to the constrainedness of the instances, mirroring some of the intuition that lies behind successful variable ordering heuristics for search which prefer more constrained instances, and prefer higher degree variables.

For the CSP features the average predicate shape and arity are important as

is the maximum arity (CSP_PERTEN_AVG_PREDSHAPE_DESC, CSP_MAX_ARITY, CSP_PERTEN_AVG_PREDARITY). Again visual inspection of the orderings produced by these features shows that they tend to group instances from similar domains together. This, of course, might simply be a consequence of how these problems are modelled.

Looking at the worst ten features to order on we see that the clause graph features relating to the clustering coefficient perform poorly (SUPPORT_CLUSTER-COEFF-MAX_DESC, SUPPORT_CLUSTER-COEFF-COEFF-VARIATION_DESC, DIRECT_CLUSTER-COEFF-MEAN_DESC, SUPPORT_CLUSTER-COEFF-MEAN_DESC). Two features relating to proximity to Horn formula also appear; the mean and max number of occurrences in a Horn clause for each variable (DIRECT_HORNY-VAR-MEAN, DIRECTORDER_HORNY-VAR-MAX). Similar to the point made previously, the poor performance from using these features reflects the poor performance associated with variable ordering anti-heuristics that prefer less constrained instances.

5 Experiments on Non-fixed Solver Configurations

These experiments involve running ReACTR in full which constantly adds and removes configurations to the configuration pool. This differs substantially from the static case we studied above, where a fixed set of solvers were constantly being selected from. Due to the lengthy solving time incurred by full ReACTR runs the dynamic experiments are limited to using a single selection policy. We use the candidate selection policy used in the feature ordering experiments (two TrueSkill, two Last Winners and two Random) for the same reasons outlined previously. The overall objective of these runs is to reduce the mean solving time.

Initially the effect of grouping on solving time was examined. The combinatorial auctions benchmark is an amalgamation of four different types of combinatorial auction instances. By grouped we mean that instances are organised by class, and all instances are kept together though within the group they may be ordered differently (Shuffled, Easy-to-Hard, Hard-to-Easy).

Figure 8 shows a scatter plot of grouped vs. ungrouped instances solving times for various orderings. All points occur above the identity line which means that ungrouped performance exceeds that of grouped in every case. This result is somewhat unexpected especially considering that the lexicographically-ordered instances (a type of grouping) had the lowest solving time in the static experiments. One possible explanation for this is that a type of over fitting occurs when the configurator only encounters instances of a single type. Due to the fact that all instances are of a single type initially, specialised configurations may beat more generally applicable configurations and cause them to be removed. This theory is supported by the trajectory of the grouped and shuffled instances in Figure 10. Here, the solving time for the initial group looks promising with

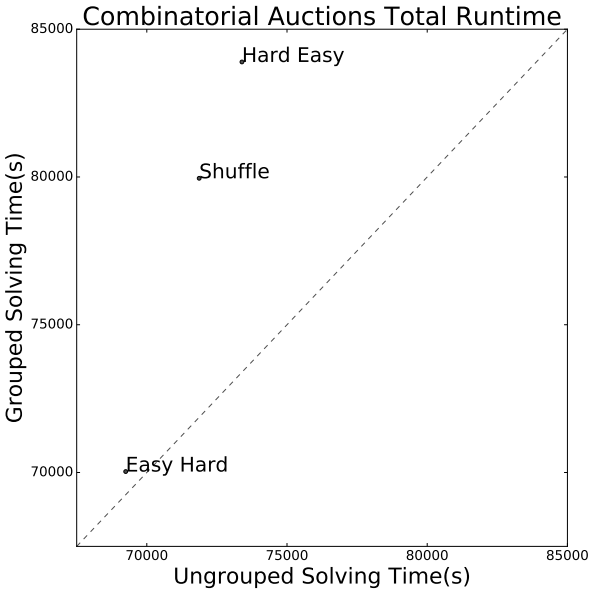


Figure 8: A scatter plot showing the total solving time for grouped vs. ungrouped instances on the Combinatorial Auctions benchmark (Avg. over 10 ReACTR executions).

a steep incline in the plot but after changing groups at 200 instances the slope becomes flatter, denoting slower solving time. Student's T-Test shows that both the Ungrouped Hard-to-Easy and Shuffled orderings outperform there Grouped counterparts (P=0.006 and P=0.054 respectively). The Easy-to-Hard results could not be shown to be statistically better (though this could be due to the relatively number of runs).

The box plots in Figure 9 provide further evidence that grouping is not beneficial during dynamic ReACTR runs. Interestingly, not only do all grouped runs perform worse than their ungrouped counterparts but they also exhibit a much larger spread in terms of solving time. Figure 9 also shows that ordering instances by Easy-to-Hard results in the fastest solving time regardless of grouping or not; Shuffled instances are in the middle in terms of solving time, and Hard-to-Easy instances take the longest to solve. This agrees with what would be expected intuitively. At the start the configurator has not had a chance to find any improvement and so solving hard problems is detrimental to the overall solving time. Figures 10 and 11 show what is happening more clearly; Hard-to-Easy solves fewer instances early on and, as such, the configurator learns less while also solving the instances more slowly because the configurations have not improved yet. Easy-to-Hard is able to solve the easy instances first which even the default configuration, which is used due to warm starting, should solve relatively quickly. Because the instances at the beginning of the Easy-to-Hard

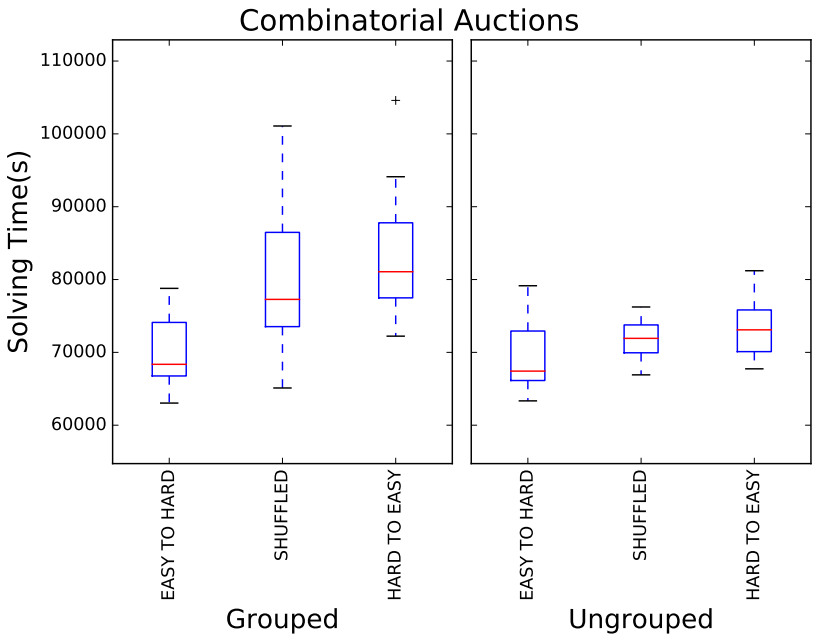


Figure 9: The total solving time for various orderings of the Combinatorial Auctions benchmark both grouped and ungrouped (Avg. over 10 ReACTR executions).

ordering are less challenging, ReACTR is also able to solve more in a short amount of time. This allows the configurator to learn better configurations much faster than if it were solving hard instances. By the time the solver has reached the more difficult instances at the end of the Easy-to-Hard order of instances it has learned multiple good configurations to make solving difficult instances much quicker.

6 Conclusions and Future Work

This paper has investigated what affect instance ordering and candidate selection has on the real-time algorithm configurator ReACTR. We demonstrate that both the selection procedure used to select configurations and the order which instances arrive in have a significant impact the ReACTR's performance.

Furthermore we show that instance ordering and the selection procedure used are linked. Some candidate selection procedures are better suited to certain instance orderings than others. Choosing the correct selection procedure can lead to marked improvements in configurator performance.

The efficiency of ReACTR when configuring over streams various instance orderings is also examined. Using two static datasets (with no configuration

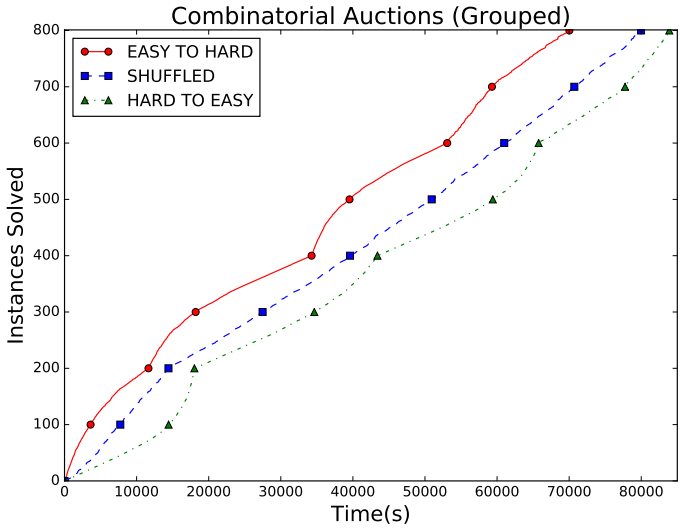


Figure 10: Instances Solved vs. Solving Time for grouped instances of the Combinatorial Auctions benchmark (Avg. over 10 ReACTR executions).

replacement occurs) in different domains we show that ordering based on nearly any feature value is better than doing so at random. We also evaluate difficulty and group based orderings using full ReACTR configuration runs. Somewhat surprisingly in this case grouping appears to hamper the performance of the configurator. This is believed to be due to overfitting that occurs when the configurator encounters only one type of instance for a long time and discards more generally applicable configurations from its pool.

Future work will focus on preventing overfitting by detecting when a change occurs in grouped instance and re-evaluating previous incumbents. Another aspect of future work will be to consider automatically detecting the incoming instance ordering and choosing the most appropriate candidate selection procedure for this. We also aim to investigate whether processing smaller batches of instances in a specific order can result in performance improvements. For example if there is a backlog of instances is it advantageous to process these based on a particular feature ordering? Finally, we will consider optimising other statistics related to solving streams of instances arriving online.

Acknowledgments

This publication has emanated from research conducted with the financial support of Science Foundation Ireland (SFI) under Grant Number SFI/12/RC/2289, co-funded under the European Regional Development Fund.

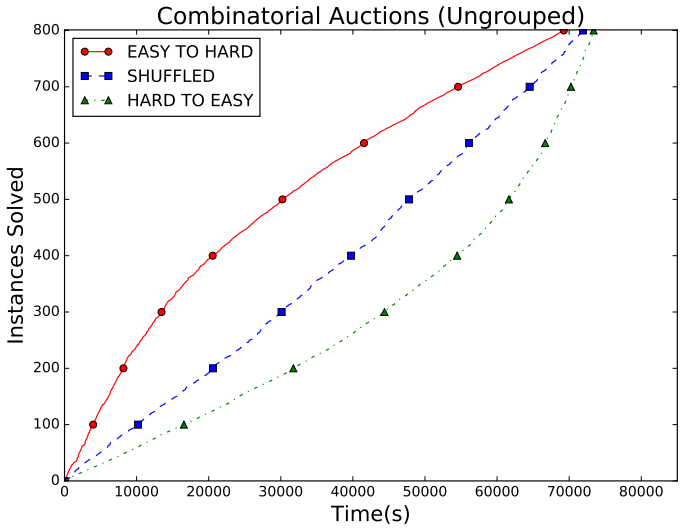


Figure 11: Instances Solved vs. Solving Time for ungrouped instances of the Combinatorial Auctions benchmark (Avg. over 10 ReACTR executions).

References

[1] Hutter F, Hoos HH, Leyton-Brown K. Sequential model-based optimization for general algorithm configuration. In: Proceedings of LION, pp. 507–523. Springer, 2011.

[2] Hutter F, Lindauer M, Balint A, Bayless S, Hoos H, Leyton-Brown K. The configurable SAT solver challenge (CSSC). *Artificial Intelligence*, 2017. **243**:1–25.

[3] Bartz-Beielstein T, Lasarczyk CW, Preuß M. Sequential parameter optimization. In: Evolutionary Computation, 2005. The 2005 IEEE Congress on, volume 1. IEEE, 2005 pp. 773–780.

[4] Hutter F, Hoos HH, Leyton-Brown K, Stützle T. ParamILS: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 2009. **36**(1):267–306.

[5] Fitzgerald T, Malitsky Y, O'Sullivan B. Reactr: Realtime algorithm configuration through tournament rankings. In: Proceedings of IJCAI. 2015 .

[6] Birattari M, Stützle T, Paquete L, Varrentrapp K, et al. A Racing Algorithm for Configuring Metaheuristics. In: GECCO, volume 2. 2002 pp. 11–18.

[7] López-Ibáñez M, Dubois-Lacoste J, Cáceres LP, Birattari M, Stützle T. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 2016. **3**:43–58.

[8] Ansótegui C, Sellmann M, Tierney K. A gender-based genetic algorithm for the automatic configuration of algorithms. In: Proceedings of CP 2009, pp. 142–157. Springer, 2009.

[9] Kadioglu S, Malitsky Y, Sellmann M, Tierney K. ISAC-Instance-Specific Algorithm Configuration. In: ECAI, volume 215. 2010 pp. 751–756.

[10] Arbelaez A, Hamadi Y, Sebag M. Continuous search in constraint programming. In: Autonomous Search, pp. 219–243. Springer, 2011.

[11] Fitzgerald T, Malitsky Y, O'Sullivan B, Tierney K. React: Real-time algorithm configuration through tournaments. In: Proceedings of SOCS. 2014 .

[12] Degroote H, Bischl B, Kotthoff L, De Causmaecker P. Reinforcement Learning for Automatic Online Algorithm Selection-an Empirical Study. In: ITAT 2016 Proceedings, volume 1649. 2016 pp. 93–101.

[13] Degroote H. Online Algorithm Selection. In: Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017. 2017 pp. 5173–5174.

[14] Bengio Y, Louradour J, Collobert R, Weston J. Curriculum learning. In: Proceedings of the 26th annual international conference on machine learning. ACM, 2009 pp. 41–48.

[15] Styles J, Hoos H. Ordered racing protocols for automatically configuring algorithms for scaling performance. In: Proceedings of the 15th annual conference on Genetic and evolutionary computation. ACM, 2013 pp. 551–558.

[16] Schneider M, Hoos H. Quantifying homogeneity of instance sets for algorithm configuration. *Learning and Intelligent Optimization*, 2012. pp. 190–204.

[17] Malitsky Y, Sabharwal A, Samulowitz H, Sellmann M. Algorithm Portfolios Based on Cost-Sensitive Hierarchical Clustering. In: IJCAI, volume 13. 2013 pp. 608–614.

[18] Hurley B, Kotthoff L, Malitsky Y, O'Sullivan B. Proteus: A Hierarchical Portfolio of Solvers and Transformations. In: Proceedings of CPAIOR. 2014 pp. 301–317.

[19] Xu L, Hutter F, Hoos HH, Leyton-Brown K. SATzilla: portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 2008. pp. 565–606.

[20] Herbrich R, Minka T, Graepel T. Trueskill: A Bayesian skill rating system. In: Advances in neural information processing systems. 2006 pp. 569–576.

[21] Vermorel J, Mohri M. Multi-armed bandit algorithms and empirical evaluation. In: Proceedings of ECML. Springer, 2005 pp. 437–448.

[22] Kuleshov V, Precup D. Algorithms for multi-armed bandit problems. *arXiv preprint arXiv:1402.6028*, 2014.

[23] Balint A, Belov A, Diepold D, Gerber S, Järvisalo M, Sinz C (eds.). Proceedings of SAT Challenge 2012. 2012.

[24] Bischl B, Kerschke P, Kotthoff L, Lindauer M, Malitsky Y, Fréchette A, Hoos H, Hutter F, Leyton-Brown K, Tierney K, et al. Aslib: A benchmark library for algorithm selection. *Artificial Intelligence*, 2016. **237**:41–58.

[25] Biere A. Lingeling. *SAT Race*, 2010.

[26] Hutter F, López-Ibáñez M, Fawcett C, Lindauer M, Hoos HH, Leyton-Brown K, Stützle T. AClib: A benchmark library for algorithm configuration. In: Proceedings of LION. Springer, 2014 pp. 36–40.

[27] Leyton-Brown K, Pearson M, Shoham Y. Towards a universal test suite for combinatorial auction algorithms. In: Proceedings ACM-EC. ACM, 2000 pp. 66–76.

[28] IBM, 2014. IBM ILOG CPLEX Optimization Studio 12.6.1.

[29] Xu L, Hutter F, Hoos HH, Leyton-Brown K. SATzilla2009: an automatic algorithm portfolio for SAT. *SAT*, 2009. **4**:53–55.

[30] Xu L, Hutter F, Hoos H, Leyton-Brown K. Features for SAT. *University of British Columbia,, Tech. Rep*, 2012.

[31] Hurley B, Kotthoff L, Malitsky Y, OSullivan B. Proteus: A hierarchical portfolio of solvers and transformations. In: International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems. Springer, 2014 pp. 301–317.

[32] OMahony E, Hebrard E, Holland A, Nugent C, OSullivan B. Using case-based reasoning in an algorithm portfolio for constraint solving. In: Irish conference on artificial intelligence and cognitive science. 2008 pp. 210–216.

[33] Zongker D. trueskill.py, 2014. Https://github.com/dougz/trueskill.

[34] Malitsky Y, Sellmann M. Instance-specific algorithm configuration as a method for non-model-based portfolio generation. In: Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems, pp. 244–259. Springer, 2012.

[35] UBC, 2013. Configurable SAT Solver Competition, URL http://www.cs.ubc.ca/labs/beta/Projects/CSSC2013/.

[36] Brummayer R, Lonsing F, Biere A. Automated testing and debugging of SAT and QBF solvers. In: Theory and Applications of Satisfiability Testing–SAT 2010, pp. 44–57. Springer, 2010.

[37] Birattari M, Yuan Z, Balaprakash P, Stützle T. F-Race and iterated F-Race: An overview. In: Experimental methods for the analysis of optimization algorithms, pp. 311–336. Springer, 2010.

[38] Fitzgerald T, O'Sullivan B. Analysing the effect of candidate selection and instance ordering in a realtime algorithm configuration system. In: Proceedings of the Symposium on Applied Computing. ACM, 2017 pp. 1003–1008.