ReACT: Real-Time Algorithm Configuration through Tournaments

Tadhg Fitzgerald and **Yuri Malitsky** and **Barry O'Sullivan**
Insight Centre for Data Analytics
Department of Computer Science, University College Cork, Ireland
(tadhg.fitzgerald, yuri.malitsky, barry.osullivan)@insight-centre.org

Kevin Tierney
Department of Business Information Systems
University of Paderborn, Germany
tierney@dsor.de

Abstract

The success or failure of a solver is oftentimes closely tied to the proper configuration of the solver's parameters. However, tuning such parameters by hand requires expert knowledge, is time consuming, and is error-prone. In recent years, automatic algorithm configuration tools have made significant advances and can nearly always find better parameters than those found through hand tuning. However, current approaches require significant offline computational resources, and follow a train-once methodology that is unable to later adapt to changes in the type of problem solved. To this end, this paper presents Real-time Algorithm Configuration through Tournaments (ReACT), a method that does not require any offline training to perform algorithm configuration. ReACT exploits the multi-core infrastructure available on most modern machines to create a system that continuously searches for improving parameterizations, while guaranteeing a particular level of performance. The experimental results show that, despite the simplicity of the approach, ReACT quickly finds a set of parameters that is better than the default parameters and is competitive with state-of-the-art algorithm configurators.

Introduction

Algorithms often have a large number of settings and parameters that control various aspects of their behaviour. Such parameters can control characteristics of an algorithm as fine grained as a value for the learning rate, or as drastic as switching the search strategy that is to be employed. Correctly setting these values for the problem instances being solved can mean the difference of orders of magnitude in performance. The task of setting these parameters is known as algorithm configuration.

In the past, many solvers had the majority of their parameters hard coded to values set by the developers, which often hindered the potential of these solvers. After all, the developers are usually not aware of all the possible applications where the solver will be used in the future. It is also now increasingly recognized that there is no single solver or parameter setting that works best on every type of instance. Instead, different problems are best solved by different strategies, meaning solvers should open parameters to the end-user in order to achieve maximum performance (Hoos 2012).

Tuning parameters by hand requires significant amounts of time, as well as intimate domain knowledge. To further complicate matters, it is impossible to systematically test even a small percentage of possible configurations due to the large configuration space of most solvers. Recent research has therefore focused on the area of automatic algorithm configuration. These approaches generally employ a train-once view, in which a representative set of instances is used to determine good parameter settings. A variety of techniques have been proposed for algorithm configuration, including model fitting (Hutter et al. 2010), genetic algorithms (Ansotegui, Sellmann, and Tierney 2009), iterated local search (Hutter et al. 2009) and racing (Birattari 2002). These approaches have proved highly successful, sometimes gaining several orders of magnitude improvements over default parameter values (Kadioglu et al. 2010; Hutter, Hoos, and Leyton-Brown 2012; Malitsky et al. 2013).

All current automatic algorithm configuration methods, however, face a number of drawbacks, the chief of which is the requirement for the existence of a representative training set. Specifically, even though the exact implementations differ, all approaches work by trying numerous parameterization offline on the available data. Yet, such datasets are not always available, especially in industry. Consider, for example, a shipping company that must create a loading plan for containers at each port its ships travel to. Generating a good loading plan quickly requires a solver with well-tuned parameters. Since finding a good plan can reduce transport costs by thousands of dollars (Delgado, Jensen, and Schulte 2009), finding good parameters for a solver is important to the efficiency of the business. Old data may not be available or too unreliable to use to boot-strap the parameters of a system, and randomly generated instances lack the structure of real problems. Furthermore, as the shipping company purchases new, larger ships or sails to different ports, the problem they are solving changes as well. Thus, the parameters for their system must also adapt to the problem being solved.

This paper introduces Real-time Algorithm Configuration through Tournaments (ReACT). With minimal overhead, ReACT exploits the availability of multiple cores in modern CPU architectures in order to run several candidate parameterizations in parallel. With the best known parameter set among those tested, we can guarantee a certain level of performance. Furthermore, by gathering statistics about parameterizations as we are solving new instances, we are able to tune a solver in real-time. This means that instead of incurring the full training cost upfront, with minimal overhead this cost is spread across the instances as they arrive. It also means that unlike in train-once approaches, we are also able to adapt to any changes in the problems as they occur.

The remainder of the paper is broken up into five sections. We first introduce existing algorithm configuration methodologies. We then describe ReACT, followed by our experimental set-up. We subsequently demonstrate the ReACT system and provide numerical results. The final section discusses our findings and offers ideas for future work.

Related Work

We first discuss methods dealing with adaptive parameter settings during the execution of an algorithm, and then move to works attempting to optimize static settings of parameters.

Reactive search optimization methods (Battiti, Brunato, and Mascia 2008), such as the well-known reactive tabu search technique (Battiti and Tecchiolli 1994), adapt parameters online during the execution of a solver or algorithm. While this allows parameters to be customized to the specifics of particular instances, information is not learned between the solving of instances. Furthermore, the starting parameters of reactive techniques may be arbitrarily bad, and require a number of iterations before converging on good values. Thus, we view reactive search as complementary to ReACT; as good starting parameters can be provided by ReACT and then refined during search.

Early algorithm configuration approaches were only able to handle a small number of continuous parameters (Birattari 2002; Adenso-Diaz and Laguna 2006; Audet and Orban 2006) (see (Eiben and Smit 2011) for a broader overview), however, there are now several competing systems that are able to configure solvers with numerous parameters regardless of whether they are continuous, discrete or categorical.

For a small set of possible parameter configurations, F-Race (Birattari 2002) and its successor, Iterated F-Race (Birattari et al. 2010), employ 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. But the problem with this is that it prefers small parameter spaces, as larger ones would require lots of testing in the primary runs. Careful attention must also be given to how and when certain parameterizations are deemed pruneable, as this greedy selection is likely to end with a sub-optimal configuration.

The parameter tuner ParamILS (Hutter et al. 2009) is able to configure arbitrary algorithms with very large numbers of discrete parameters. The approach conducts a focused iterated local search that generally starts from the default parameters of a solver. ParamILS then enters an improvement phase using a one-exchange neighborhood in an iterative first improvement search. The local search continues until a local optimum is encountered, at which point the search is repeated from a new starting point after performing a perturbation. To avoid randomly searching the configuration space, at each iteration the local search gathers statistics on which parameters are important for finding improved settings, and focuses on assigning them first. ParamILS has been shown to be successful with a variety of solvers, such as IBM CPLEX (IBM 2011).

The gender-based genetic algorithm (GGA) (Ansotegui, Sellmann, and Tierney 2009) was introduced as a population based alternative to ParamILS. GGA uses a genetic algorithm to find a good parameter configuration, as well as a population split into two genders in order to balance exploitation and exploration of the search space. At each generation, half of the population competes on a collection of training instances. The parameter settings that yield the best overall performance mate with the non-competitive population to form part of the next generation.

Most recently, Sequential Model-based Algorithm Configuration (SMAC) (Hutter, Hoos, and Leyton-Brown 2011) was introduced. This approach generates a model over a solver's parameters to predict the likely performance if the parameters were to be used. This model can be anything from a random forest to marginal predictors. The model is used to identify aspects of the parameter space, such as which parameters are most important. Possible configurations are then generated according to the model and compete against the current incumbent, with the best configuration continuing to the next iteration.

As mentioned in the introduction, all of these approaches follow a train-once methodology. This means that prior to any tuning taking place there must be a representative dataset of instances available, something that may not always be present. Additionally, if the stream of instances changes over time, for example due to a change in conditions, a new dataset must be collected and the solver retrained. Even with only a short timeout of 5 minutes and a handful of parameters, such a task might easily require several days worth of computation. This makes these existing approaches ill-suited for continuously tuning the solver in real-time. The authors are aware of only one work, (Malitsky, Mehta, and O'Sullivan 2013), which begins to tackle this type of learning. But unlike in this work, the paper only approached the problem from an algorithm selection perspective instead of an algorithm configurator.

ReACT

This work explores what is possible in the area of real-time algorithm configuration using the power of modern multicore CPUs. The Real-time Algorithm Configuration through Tournaments methodology (ReACT) tests multiple candidate parameterizations in parallel, removing dominated parameterizations as results on more instances are observed. Algorithm 1 provides a formalization of our approach.

We initialize ReACT with the number of cores available, n; the solver to use, s; the potentially infinite sequence of instances to solve, I; the parameter weakness ratio, r; the minimum number of "wins" necessary to exclude a parameterization, m; the solver timeout, t; and an (optional) initial parameterization, p_{ws}. ReACT can be *cold started* by setting p_{ws} to *null*, in which case n random parameterizations are added to the parameter pool. When p_{ws} is provided with a parameter setting, it is added to the pool and the rest of the parameterizations are random. In this case, we say that ReACT is *warm started*. While a cold start can result in poor performance early in the search, it avoids any potential bias introduced through user provided parameters.

In order to keep track of the performance of the parameterizations, we implement a score keeping component, represented by the $n \times n$ matrix $\mathbf{S}$ on line 5. An entry $\mathbf{S}(p, p')$ represents the number of times parameterization p has had a lower runtime or better final objective than p'. For each instance in the instance sequence, ReACT runs a tournament (line 7). Parameterizations compete in a tournament by running in parallel on the given instance with a specified time-out. The first parameterization to return an optimal solution sends a termination signal to the other runs. To account for any discrepancies in start time, and thus ensuring that each approach is evaluated fairly, the termination signal also encodes the time required by the fastest parameterization to finish. When all runs have finished and terminated, the winner is the parameterization with the lowest time taken to solve the instance. In the case where no parameterizations finish in the allotted time, the winner is the parameterization which returns the best objective value. We note that unlike the GGA configurator, where a given percentage of the participants in a tournament are considered the winners, here we only consider the best parameterization as the winner, with multiple winners only in the case of a tie.

After the scores are updated for the winners of the tournament, we cull *weak* parameterizations from the pool on lines 11 and 12. We define a parameterization p as weak if it has *a*) been beaten by another parameterization, p', at least m times and *b*) the ratio of the number of times p' has beaten p to the number of times p has beaten p' is greater or equal to r. The former criteria ensures that parameterizations are not removed without being given ample opportunity to succeed. The latter criteria ensures that the domination of one parameterization over another is not just due to random chance. In our implementation of ReACT, we set $m = 10$ and $r = 2$, meaning a parameterization is weak if it has lost to another competitor at least 10 times and has lost to another parameter setting twice as many times as it has beaten that parameter setting. For example, if p beats p' ten times, but p' beats p twenty times, then p is a weak parameterization and will be removed. After removing weak parameterizations from the pool, the score keeper is updated on line 13 and new, random parameterizations are inserted into the pool.

Though we restrict the number of employed solvers to the amount of CPU cores available for our experiments, the same methodology could readily utilize nodes of a compute cluster or cloud. This would allow running a larger number of solvers simultaneously while also bypassing potential bottlenecks such as memory usage.

Algorithm 1 The ReACT algorithm

```
1:  function ReACT($n$, $s$, $I$, $r$, $m$, $t$, $p_{ws}$)
2:      $P \leftarrow \{p_{ws}\}$ if $p_{ws} \neq null$ else $\emptyset$
3:      while $|P| < n$ do
4:          $P \leftarrow P \cup \{$RND-PARAMETERIZATION()$\}$
5:      $\mathbf{S} \leftarrow 0^{n \times n}$          ▷ an $n \times n$ matrix initialized to 0
6:      for $i \in I$ do
7:          $W \leftarrow$ TOURNAMENT$(P, s, i, t)$
8:          for $w \in W$ do
9:              for $l \in 1, \ldots, n; l \notin W$ do
10:                 $\mathbf{S}(w, l) \leftarrow \mathbf{S}(w, l) + 1$
11:         $K \leftarrow \{p | \exists p' \neq p$ s.t. $\mathbf{S}(p', p) \geq m \wedge \frac{\mathbf{S}(p', p)}{\mathbf{S}(p, p')} \geq r\}$
12:         $P \leftarrow P \setminus K$
13:         $\mathbf{S}(k, p) \leftarrow 0, \mathbf{S}(p, k) \leftarrow 0,\ \forall k \in K; p \in P$
14:         for $k \in K$ do
15:             $P \leftarrow P \cup \{$RND-PARAMETERIZATION()$\}$
```

We would again like to emphasize that although the approach is relatively straight forward, the presented scoring component does provide a reasonable assurance that once a new improving parameterization is found that it will retain those settings. In particular, for every time one new parameter setting defeats another, the latter must solve two additional instances to prove its dominance over the first. This means that any new candidate must prove its merit only a few times before it is taken seriously and becomes hard to remove by the current incumbent. Yet the requirement to solve twice as many instances simultaneously ensures that it is very probable that a new parameterization will come along that does not have to spend a large number of runs to prove it is superior. This property allows the configuration to quickly adapt to changes in the observed instances while mitigating the chances we will get rid of a good parameterization on a whim.

The tournaments in ReACT are inspired from the tournaments in GGA. The tournaments serve a particularly important function in ReACT, which is that they ensure that the user is not affected by the search for new, effective parameterizations. The user receives a solution to a particular instance at the same time as new configurations are experimented with, due to the parallel nature of the approach. Since solvers using poorly performing parameterizations are terminated once a solution is found by any parameter setting in the pool, the user only has to wait as long as it takes for the best parameter setting to find an answer.

In striking a balance between intensification and diversification in its search methodology, ReACT tends towards diversity. The random parameterizations that are added to replace dominated ones provide strong diversification to avoid local optima. At first glance, simply choosing new parameterizations completely at random may seem too simple to actually function. However, there is a good reason for this randomness. As the instances change over time, diversity of parameters is critical to being able to find new parameterizations for those instances. A more greedy approach runs the risk of getting stuck in the wrong basin of attraction when the instances shift, leaving the user with a longer period of poor solving performance.

ReACT performs intensification by throwing out poorly performing parameter settings. Our strategy here is still somewhat conservative, which stands in contrast to train-once parameter tuners like GGA, which intensifies through a strict selection procedure, or ParamILS, which has a greedy iterative first improvement step. ReACT benefits from keeping parameter settings that are "good enough" around. These parameterizations could turn out to be beneficial if the instances shift in a particular direction. By keeping such parameters alive, our approach is always prepared for whatever may come next from the instance sequence. We only throw out parameters when it is clear that they are dominated by other parameter settings, although it is possible that we are sometimes unlucky and throw out a good parameter setting.

One of the main drawbacks of our strategy, is that finding good parameter settings by chance is not easy. We are aided by the fact that there tend to be regions of good parameterizations, rather than single parameter configurations that perform extremely well. By maintaining diversity, ReACT can hold on to parameters from a number of good regions and is prepared for new types of instances. Other forms of selecting new parameter settings could also be imagined, such as using a model based technique or mutation procedure of good performing parameters. We hold such techniques for future work, as ensuring the diversity of the approach under such intensification procedures is not trivial.

As presented, there are many parts of the algorithm that can be readily improved with more sophisticated techniques. However, the goal of this paper is to introduce the new ReACT methodology, leaving other comparisons as potential future work. We also emphasize that even with these preliminary techniques, as will be shown in subsequent sections, ReACT is able to significantly outperform the default parameters and achieve a level of performance on par with the state-of-the-art train-once methodology at a far smaller cost.

Experimental Setup

In order to properly test ReACT, we require a dataset with several properties. First, the instances must be relatively homogeneous, i.e., the instances are all of a similar type or variant of a problem. Without homogeneity, we cannot assume that a single parameter set can work well across all of the instances. Heterogeneous datasets are best handled with algorithm selection techniques and we leave this for future work. Second, the instances must be hard, but not too hard. Instances requiring too much time to solve will timeout, thereby offering no information to an algorithm configurator about which parameter setting is best. This results in the configurator performing a random walk. Meanwhile, datasets that are too easy will not provide noticeable gains.

For the experiments we focused on combinatorial auction problems encoded as mixed integer problems, as generated by the Combinatorial Auction Test Suite (CATS) (Leyton-Brown, Pearson, and Shoham 2000). Using CATS, we created datasets consisting of a homogeneous set of instances that are solvable in a reasonable amount of time. CATS has a number of different options for generating problem instances, including the number of bidders and goods in each instance, as well as the type of combinatorial auction.

We focused on two types of auction (regions and arbitrary) in order to provide homogeneous datasets for our tuner to work with. The regions problem class is meant to simulate situations where proximity in space matters. This is typically the case when plots of land are auctioned, where its a lot easier for a company to develop neighboring properties rather than widely separated ones. Alternatively, the arbitrary class is designed so that there is not a determined relation between various goods, as is likely to be the case when dealing with collector items. While both datasets deal with combinatorial auctions, the structures of the corresponding instances are very different and therefore lend themselves to different solution strategies. The reader is referred to the original CATS algorithm description (Leyton-Brown, Pearson, and Shoham 2000) for more information about the instances.

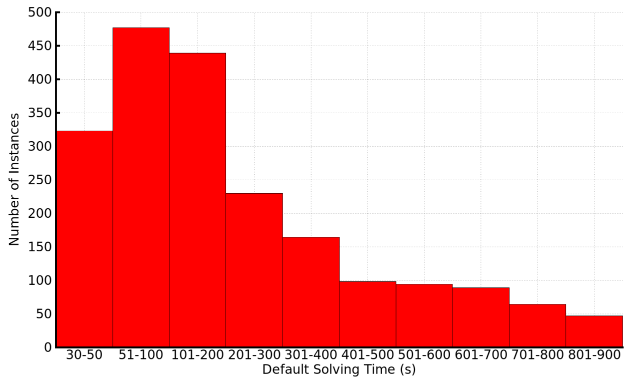
For the regions dataset, we generate our instances with 250 goods (standard deviation 100) and 2000 bids (standard deviation 2000). Most of the instances we generated can be solved in under 900 seconds. Combining this with a solver timeout of 500 seconds means that some of the instances are challenging, but can be potentially solved with slightly better parameters within the time limit. The arbitrary instances were generated with 800 goods (standard deviation 400) and 400 bids (standard deviation of 200). These experiments maintained the 500 second timeout.

We solve the instances in our dataset with IBM CPLEX (IBM 2011). CPLEX is a state-of-the-art mathematical programming solver used widely both in industry and academia. The solver has over 100 adjustable parameters that govern its behaviour, and has been shown in the past to be rather amiable to automated configuration (Kadioglu et al. 2010; Hutter, Hoos, and Leyton-Brown 2011).

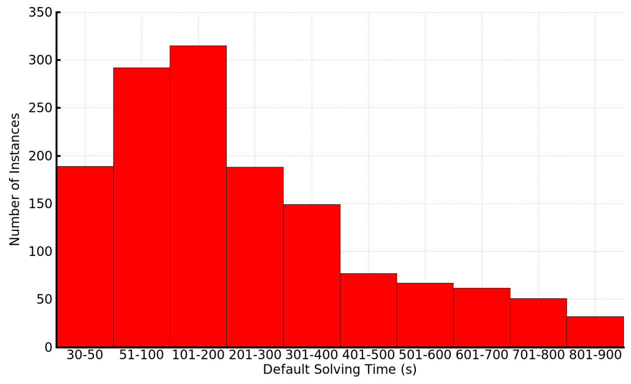
When generating our benchmark dataset, we removed all instances that could be solved in under 30 seconds using the default CPLEX parameter settings. These instances are too easy to solve, meaning that they would just introduce noise into our configuration procedure, as it can be somewhat random which parameter setting solves them first. We stress that removing these instances can be justified in the presence of a straightforward pre-solver, a practice commonly used for algorithm selection techniques prior to predicting the best solver for an instance. Our final regions dataset is comprised of 2,000 instances whose difficulty, based on performance of the default parameters, was mainly distributed between 30 and 700 seconds, but with all instances being solvable in under 900 seconds, as shown by Figure 1a. Meanwhile, the final arbitrary dataset was comprised of 1,422 instances, whose distribution of runtimes with default parameters can be seen in Figure 1b.

Results

We test our methodology on three different scenarios on each of the two combinatorial auction datasets. In the first, we assume that instances are being processed at random, so we shuffle all of our data and feed it to our tuner one at a time. We also try two variations where problems change



(a) Regions



(b) Arbitrary

Figure 1: The distribution of instances in the final dataset based on solving time using the CPLEX default parameters.

steadily over time. In the first case, we assume that the auction house grows and is able to take in larger inventories, so the number of goods in each new instance is monotonically increasing. In the second case, we create a scenario where the auction house becomes more successful, thus each new problem instances has a monotonically increasing number of bids. Regardless of the scenario, however, each instance is given a timeout of 500 seconds on a two Intel Xeon E5430 Processors (2.66 GHz) with eight cores. For ReACT, though, we restrict ourselves to using only six cores to avoid running out of memory.

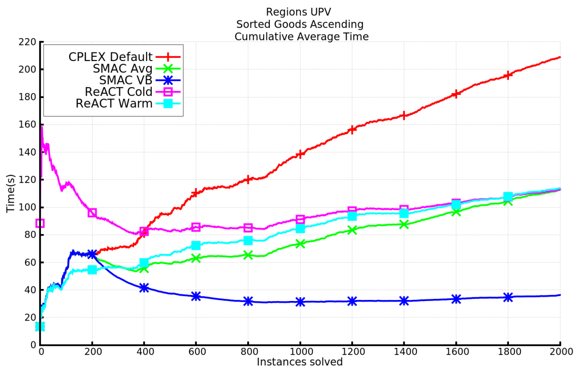
As a comparison, in addition to the default parameters, we compare ReACT with the state-of-the-art train-once approach SMAC. Here we simulate the case where initially no training instances exist and we must use the default parameters to solve the first 200 instances. SMAC then uses the first 200 instances as its training set and is tuned for a total of two days. After those 48 hours, each new instance is solved with the tuned parameters. In all presented results we follow the generally accepted practice of tuning multiple versions with SMAC and selecting the best one based on the training performance. In our case we present both the result of the best found SMAC parameters, and the average performance of the six parameterizations we trained. We refer to these as SMAC-VB and SMAC-Avg, respectively. Note that in this case SMAC-VB is equivalent to running all 6 tuned versions of SMAC in parallel on the same number of cores as are available to ReACT.

For our evaluations, we compare two versions of ReACT. In the first, we assume that no information is known about the solver beforehand, and thus all the initial parameterizations are generated at random. We refer to this case as ReACT-cold. We also test ReACT-warm, where one of the initial parameterizations contains the default settings.

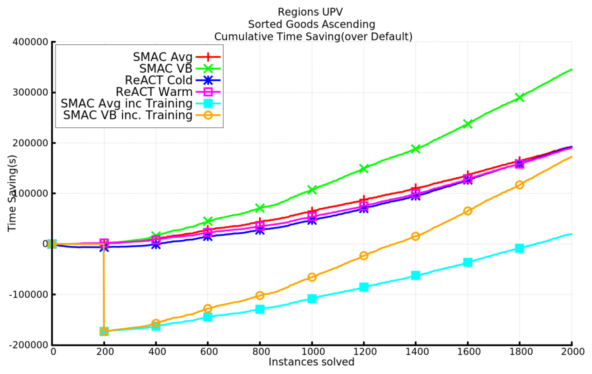
To avoid presenting a result due to a lucky starting parameterizations or random seed, we run each version of ReACT three times, and present the average of all three runs in the plots and tables of this section. Figure 2 summarizes the results on the regions datasets. In all these plots, first note that Figure 2a presents the cumulative average time per instance for the scenario, where the number of goods continuously increases with each instance. Note that the default curve rises steadily with each new instance. This is because as the number of goods increases, the difficulty of the problem also increases, and the default parameters are unable to adapt. Alternatively, notice that ReACT-warm, which initially has the default parameters as one of its starting parameters, is able to adjust to the change and achieve a significant improvement after only 150 instances. Even when the initial parameters are chosen at random, ReACT-cold is able to quickly achieve a level of performance such that the cumulative average outperforms the default in only 400 instances.

Figures 2c and 2e tell a similar story. In each case within observing 200 to 400 instances either version of ReACT is able to overtake the performance of default CPLEX parameters. Studies on the median runtime of instances follow a very similar trend to the curves in Figures 2a, 2c, and 2e. The fact that ReACT comes so close to the performance of SMAC with such a simple approach, in which the next parameterization is selected completely at random, bodes well for future iterations of real-time tuning.

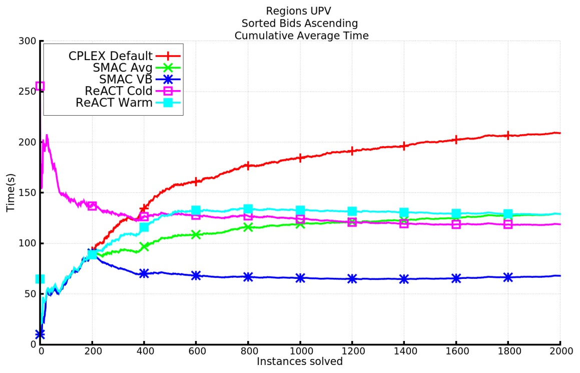
The true benefit of ReACT is visible clearly in Figures 2b, 2d, and 2f. These figures present the cumulative amount of time saved by the associated methodology over using the default parameters. After observing the corresponding cumulative average time, it is no surprise that the savings continuously improve with each newly observed instance. What we also observe at first glance is that the parameters found by ReACT perform as well as those for a typical result after a 2-day SMAC run. And ReACT seems not that much worse than the SMAC-VB. This observation, however, is a bit misleading. Recall that in order to find the SMAC parameters, a 2-day training period is essential. During this time a company would be forced to use the default parameters, which clearly do not scale. If we consider that on average instances arrive continuously, in the time it took to train SMAC, another 800 instances would have gone by. However, with ReACT, good parameters are found throughout the testing period. In fact, we are able to definitively tune over 60 parameters in real-time with minimal overhead and achieve a per-



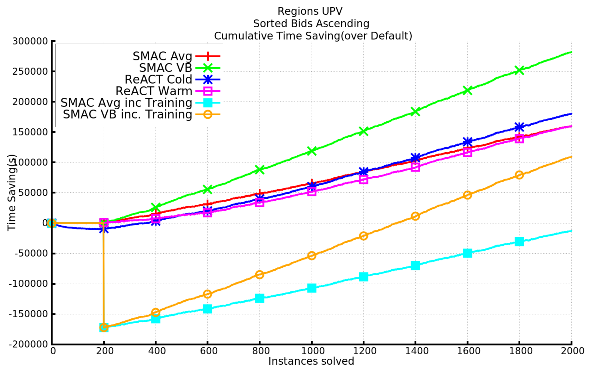
(a) Cumulative average runtime on dataset with number of goods monotonically increasing.



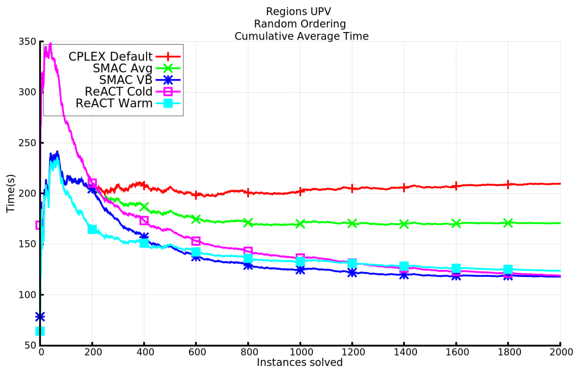
(b) Cumulative time savings on dataset with number of goods monotonically increasing.



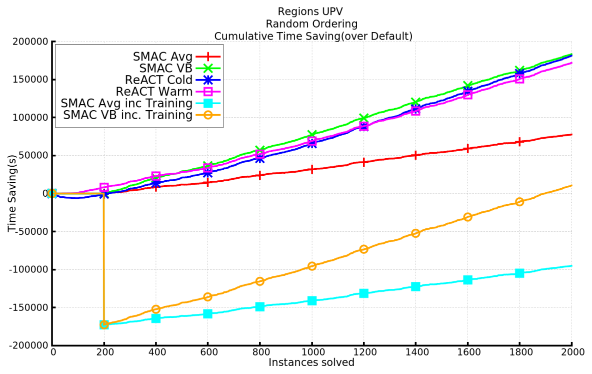
(c) Cumulative average runtime on dataset with number of bids monotonically increasing.



(d) Cumulative time savings on dataset with number of bids monotonically increasing.

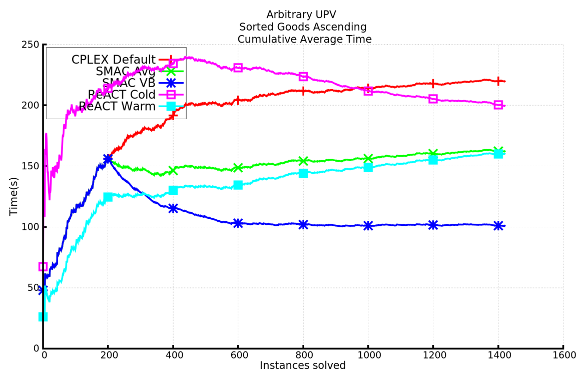


(e) Cumulative average runtime on dataset with random ordering of instances.

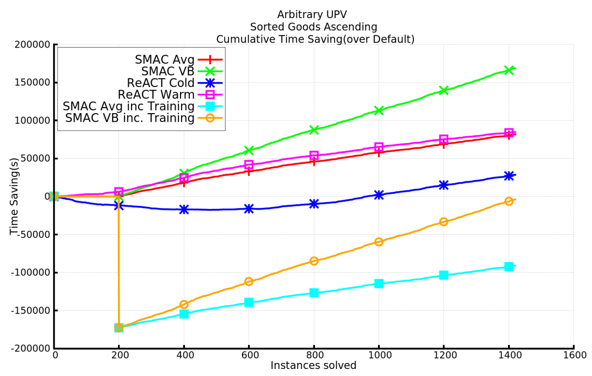


(f) Cumulative time savings on dataset with random ordering of instances.

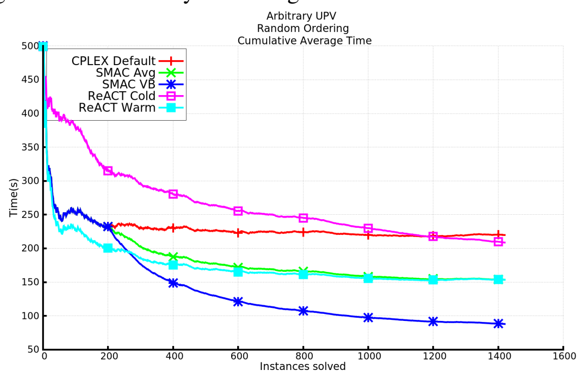
Figure 2: Cumulative average runtime and cumulative savings of techniques on the three permutations of the regions dataset. On all plots, the x-axis specifies the total number of observed instances. For (a), (c), and (e), the y-axis specifies the solution time in seconds. The plots (b), (d), and (f) show the time saved over the default parameterization of CPLEX, with the y-axis being the number of seconds saved so far.



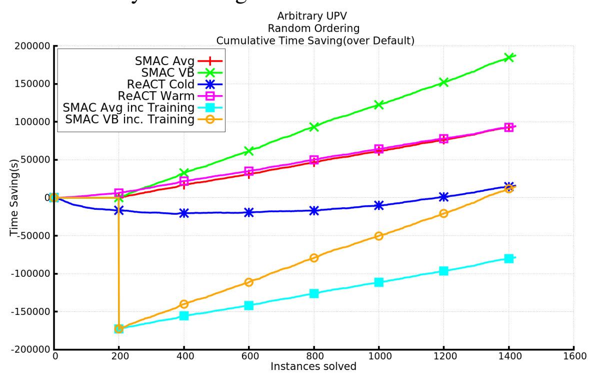
(a) Cumulative average runtime on dataset with number of goods monotonically increasing.



(b) Cumulative time savings on dataset with number of goods monotonically increasing.



(c) Cumulative average runtime on dataset with random ordering of instances.



(d) Cumulative time savings on dataset with random ordering of instances.

Figure 3: Cumulative average runtime and cumulative savings of techniques on the arbitrary dataset. On all plots, the x-axis specifies the total number of observed instances. For (a) and (c) the y-axis specifies time in seconds. Plots (b) and (d) show time saved over the default parameterization of CPLEX, with the y-axis being the number of seconds saved so far.

formance drastically better than default. We also achieve a performance on par with the average SMAC run.

Furthermore, let's hypothetically assume that after we observe the first 200 instances, no new instances arrive for the next two days. This is an unlikely scenario, but it allows us to definitively show the computational cost of the training in Figures 2b, 2d, and 2f in the form of SMAC VB-inc and SMAC Avg-inc. Here, we clearly see that even after observing the remaining 1,800 instances, parameters found by SMAC don't offset the upfront cost of finding them. ReACT though, finds its parameters with no temporal overhead.

Note also that this means that although SMAC VB-inc appears to steadily converge on ReACT, it may not be indicative of the long term behavior. ReACT continuously discovers better parameters as it progresses and is likely to eventually find parameters better than those of SMAC while the parameters which SMAC uses remain unchanged.

Additionally, it cannot be said that SMAC is not using as much resources as ReACT. As was stated in its original paper, SMAC is known to not always return a good parameterization. To address this, several versions have to be tuned in parallel, with the best performing one on validation instances being the finally selected set. Thus, training SMAC takes as many cores as are currently utilized by ReACT.

Figure 3 shows the plots on arbitrary auctions for a random ordering of incoming instances and the scenario where the number of goods increases with each subsequent instance. Due to space constraints the bids sorted plot is omitted, however, it exhibits similar behaviour to that of the goods ordered dataset. Note that the cold-started ReACT is slower in finding a parameter set that outperforms the default, yet even on these harder instances the gains become evident. And just like in the regions scenarios, the upfront tuning costs necessitated by SMAC are hard to overcome.

As an alternative view of the results, Table 1 presents the amount of cumulative time saved by each tuning approach over the default parameters. The numbers are in thousands of seconds. In all cases, ReACT is able to outperform default parameters and is on a par with SMAC Avg. Given the fact that the CPLEX solver has been hand tuned by experts for over a decade and SMAC is the current state of the art, this is an encouraging feat. Interestingly, for the case where the new regions instances arrive in a random order, ReACT overtakes the performance of the average SMAC parameter

Table 1: Cumulative amount of time saved over the default CPLEX parameters in thousands of seconds after n instances have been solved. Three orderings of the regions and arbitrary datasets are shown: random, in which instances are processed in a random order; sorted goods, where the number of goods in each subsequent instance is monotonically increasing; and sorted bids, in which the number of bids of each new instance is monotonically increasing.

Regions Random	After n Instances			
	500	1000	1500	2000
SMAC Avg.	12	31	54	78
SMAC Virtual Best	29	77	131	183
SMAC Virtual Best (inc.)	-144	-96	-42	11
ReACT Cold-start	21	66	122	181
ReACT Warm-start	28	69	118	172

Arbitrary Random	After n Instances			
	350	700	1050	1400
SMAC Avg.	12	39	65	93
SMAC Virtual Best	24	78	130	185
SMAC Virtual Best (inc.)	-149	-95	-43	12
ReACT Cold-start	-20	-17	-7	15
ReACT Warm-start	17	43	68	93

Regions Goods Sorted (Asc)	After n Instances			
	500	1000	1500	2000
SMAC Avg.	18	65	123	192
SMAC Virtual Best	29	106	212	345
SMAC Virtual Best (inc.)	-144	-66	39	172
ReACT Cold-start	6	47	110	192
ReACT Warm-start	15	54	113	190

Arbitrary Goods Sorted (Asc)	After n Instances			
	350	700	1050	1400
SMAC Avg.	13	40	60	81
SMAC Virtual Best	22	74	120	166
SMAC Virtual Best (inc.)	-151	-98	-53	-7
ReACT Cold-start	-17	-14	5	27
ReACT Warm-start	20	48	68	84

Regions Bids Sorted (Asc)	After n Instances			
	500	1000	1500	2000
SMAC Avg.	24	65	113	160
SMAC Virtual Best	42	118	202	282
SMAC Virtual Best (inc.)	-130	-54	30	109
ReACT Cold-start	13	60	121	180
ReACT Warm-start	13	52	105	160

Arbitrary Bids Sorted (Asc)	After n Instances			
	350	700	1050	1400
SMAC Avg.	6	33	64	94
SMAC Virtual Best	15	79	158	224
SMAC Virtual Best (inc.)	-158	-93	-15	51
ReACT Cold-start	-20	-14	16	46
ReACT Warm-start	9	34	63	95

set and is close to that of SMAC VB. This is likely because the first 200 randomly sorted instances tend to be harder in general, so two days might not be enough to fully tune SMAC. However, ReACT is not affected by this as it does not require initial offline tuning.

It is interesting to note that, while we continuously observe new potential parameterizations that occasionally win the tournament, in all our scenarios there is usually one parameter set that the solver continuously comes back to two thirds of the time. This means that ReACT quickly finds a good parameterization to use as its core, while occasionally taking advantage of some parameterization getting lucky. The fact that nothing was able to kick this core one out, is also a testament that the scoring metric we use is fair yet resilient to noise. This is again confirmed by stating that for warm start ReACT, in all observed cases, the default parameters are thrown out after at most 300 instances.

Conclusion

The paper introduces a novel algorithm configuration methodology that searches for improving parameters as it solves a stream of new instances. A typical situation in the real world where solvers must continuously tackle problems as part of a decision support systems. By taking the changes in problem structure into account on the fly, Real-time Algorithm Configuration through Tournaments (ReACT) alleviates the need for the expensive offline, train-once tuning.

ReACT offers a simple way for companies or researchers to integrate algorithm configuration into their systems without needing large amounts of extra computational power. We achieve this by leveraging the multi-core architecture that is becoming prevalent in modern computers. Running several parameterizations in parallel, we can terminate poorly performing parameter configurations as soon as a solution is found. We also highlight that even though we restrict our experiments to using an eight core machine, the true power of this approach is that it will continue to improve as more cores become available.

In this paper we presented an initial approach that chooses the next parameter set to try at random, which is able to achieve marked improvements over the default parameters. As future work, there are a number of interesting and promising directions that can be pursued. For example, instead of generating the setting of a new parameterization randomly, one can employ a model-based approach similar to that of SMAC (Hutter, Hoos, and Leyton-Brown 2011), or choose a parameter setting that is maximally different from anything else that has been tried so far. Instead of terminating all runs as soon as one configuration is finished, an online tracker can dynamically determine if the current run is significantly faster than expected. If so, it can allow a small amount of additional time for the other configurations to finish, and thus gain a better view of the dominance relation between configurations. Instead of the direct factor of two measure for dominance, an alternate ratio can be used, not to mention a full statistical test. All these, and many other approaches have the potential to further strengthen our results. This paper is but the first necessary step that lays the foundation of this line of research.

Acknowledgments. This publication has emanated from research supported in part by a research grant from Science Foundation Ireland (SFI) under Grant Number SFI/12/RC/2289. It is also funded by the FP7-funded FET-Open project called Inductive Constraint Programming (ICON), Grant Agreement 284715. Finally, the authors thank the Paderborn Center for Parallel Computing for the use of their systems.

References

Adenso-Diaz, B., and Laguna, M. 2006. Fine-tuning of algorithms using fractional experimental design and local search. *Operations Research* 54(1):99–114.

Ansotegui, C.; Sellmann, M.; and Tierney, K. 2009. A gender-based genetic algorithm for the automatic configuration of algorithms. *CP* 142–157.

Audet, C., and Orban, D. 2006. Finding optimal algorithmic parameters using derivative-free optimization. *SIAM Journal on Optimization* 16(3):642.

Battiti, R., and Tecchiolli, G. 1994. The reactive tabu search. *ORSA Journal on Computing* 6(2):126–140.

Battiti, R.; Brunato, M.; and Mascia, F. 2008. *Reactive search and intelligent optimization*. Springer.

Birattari, M.; Yuan, Z.; Balaprakash, P.; and Stützle, T. 2010. F-race and iterated f-race: An overview. In Bartz-Beielstein, T.; Chiarandini, M.; Paquete, L.; and Preuss, M., eds., *Experimental Methods for the Analysis of Optimization Algorithms*. Springer. 311–336.

Birattari, M. 2002. A racing algorithm for configuring metaheuristics. In *Proceedings of the Genetic and Evolutionary Computation Conference*, 11–18. Morgan Kaufmann.

Delgado, A.; Jensen, R. M.; and Schulte, C. 2009. Generating optimal stowage plans for container vessel bays. In *CP*. Springer. 6–20.

Eiben, A. E., and Smit, S. K. 2011. Parameter tuning for configuring and analyzing evolutionary algorithms. *Swarm and Evolutionary Computation* 1(1):19–31.

Hoos, H. 2012. Programming by optimization. *Communications of the ACM* 55(2):70–80.

Hutter, F.; Hoos, H.; Leyton-Brown, K.; and Stuetzle, T. 2009. Paramils: An automatic algorithm configuration framework. *JAIR* 36:267–306.

Hutter, F.; Hoos, H.; Leyton-Brown, K.; and Murphy, K. 2010. Time-bounded sequential parameter optimization. In *LION*, 281–298.

Hutter, F.; Hoos, H.; and Leyton-Brown, K. 2011. Sequential model-based optimization for general algorithm configuration. In *Learning and Intelligent Optimization*. Springer. 507–523.

Hutter, F.; Hoos, H.; and Leyton-Brown, K. 2012. Parallel algorithm configuration. In *LION*.

IBM. 2011. IBM ILOG CPLEX Optimization Studio 12.5.

Kadioglu, S.; Malitsky, Y.; Sellmann, M.; and Tierney, K. 2010. ISAC – Instance-Specific Algorithm Configuration. *ECAI* 751–756.

Leyton-Brown, K.; Pearson, M.; and Shoham, Y. 2000. Towards a universal test suite for combinatorial auction algorithms. In *Proceedings of the 2nd ACM conference on Electronic commerce*, 66–76. ACM.

Malitsky, Y.; Mehta, D.; O'Sullivan, B.; and Simonis, H. 2013. Tuning parameters of large neighborhood search for the machine reassignment problem. In *CPAIOR*.

Malitsky, Y.; Mehta, D.; and O'Sullivan, B. 2013. Evolving instance specific algorithm configuration. In *SOCS*.