ReACTR: Realtime Algorithm Configuration through Tournament Rankings

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

Yuri Malitsky
IBM T.J. Watson Research
Centre, New York, USA
ymalits@us.ibm.com

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

Abstract

It is now readily accepted that automated algorithm configuration is a necessity for ensuring optimized performance of solvers on a particular problem domain. Even the best developers who have carefully designed their solver are not always able to manually find the best parameter settings for it. Yet, the opportunity for improving performance has been repeatedly demonstrated by configuration tools like ParamILS, SMAC, and GGA. However, all these techniques currently assume a static environment, where demonstrative instances are procured beforehand, potentially unlimited time is provided to adequately search the parameter space, and the solver would never need to be retrained. This is not always the case in practice. The ReACT system, proposed in 2014, demonstrated that a solver could be configured during runtime as new instances arrive in a steady stream. This paper further develops that approach and shows how a ranking scheme, like TrueSkill, can further improve the configurator's performance, making it able to quickly find good parameterizations without adding any overhead on the time needed to solve any new instance, and then continuously improve as new instances are evaluated. The enhancements to ReACT that we present enable us to even outperform existing static configurators like SMAC in a non-dynamic setting.

1 Introduction

Automated algorithm configuration is the task of automatically finding parameter settings for a tunable algorithm which improve its performance. It is an essential tool for anyone wishing to maximize the performance of an existing algorithm without modifying the underlying approach.

Modern work on algorithm portfolios [Malitsky *et al.*, 2013b; Xu *et al.*, 2008; Malitsky and Sellmann, 2012] tells us that there is often no single approach that will produce optimal performance (solving time, solution quality etc.) in every situation. This is why modern developers, not knowing all the conditions and scenarios where their work will be employed, leave many of the parameters of their algorithms open to the user. As a side-effect of this, however, solvers might have hundreds of parameters that can be changed to fine tune their behavior to a specific benchmark. Even for solvers with a handful of parameters, it is difficult to search through all the, possibly non-linear, relations between parameters.

To alleviate the difficult and time consuming process of manually fiddling with parameters, a number of tools have been recently introduced. ParamILS [Hutter *et al.*, 2009], for example, employs an iterated local search to explore the parameter space, focussing on areas where it found improvements in the past. Alternatively, SMAC [Hutter *et al.*, 2011] tries to build an internal model that predicts the performance of a parameterization, trying the ones most probable to improve upon the current behavior. Finally, GGA [Ansótegui *et al.*, 2009] utilizes a genetic approach, competing a number of parameterizations in parallel and allowing the best ones to pass on their parameter settings to the subsequent generation. While there is no consensus on which of these approaches is best, all of them have been repeatedly validated in practice, sometimes leading to orders of magnitude improvements over what was found by human experts [Hutter *et al.*, 2011].

Despite their undeniable success, all existing configurators take a static view of the problem. They assume a train-once scenario, where a rich benchmark set of instances exists and significant time can be spent offline searching for the best parameterizations. Furthermore, they assume that once a good parameterization is found, it will be utilized without modification, forever. While applicable in many situations, there are cases when these three assumptions do not hold. Imagine, for example, the case of repeated combinatorial auctions like those commonly utilized for placing ads on webpages. New companies, ads, and keywords are constantly being introduced, which means that the size and number of goods are constantly in flux. Furthermore, new instances arrive in a continuous stream. This means that the benchmark instances are always changing, there is no time to train offline, and the problems we are solving change, meaning that the parameters need to be constantly updated. Existing configuration approaches are ill-equipped to deal with this setting.

Some work exists on online learning in the closely related area of algorithm portfolios. SUNNY: a Lazy Portfolio Approach for Constraint Solving [Amadini *et al.*, 2014] builds a schedule of CSP solvers in a portfolio online without any prior training. SUNNY finds similar instances to the current instance using k-Nearest Neighbours. It then selects a minimum subset of solvers that could the solve the greatest number of neighbouring instances and schedules them based the number of neighbouring instances solved. Alternatively, Evolving Instance-Specific Algorithm Configuration [Malitsky *et al.*, 2013a] is a portfolio approach that performs some offline learning but is able to evolve the portfolio it uses as it processes a stream of instances in order to adapt to changing instances. It does this by dynamically re-clustering incoming instances and assigning a set of solvers to each cluster.

ReACT, Realtime Algorithm Configuration through Tournaments, provides a way to perform online algorithm configuration without prior learning [Fitzgerald *et al.*, 2014]. The idea was that once a new instance arrived, n parameterizations of a solver would attempt to solve it in parallel. Once the instance was solved, all other solvers were terminated and the winner's score was upgraded. Because the current best parameterization was always among the n versions evaluated, a continuously improving bound on performance could be guaranteed without offline training. Furthermore, once a parameterization failed to win a certain percentage of the time, it was discarded, with a new random one taking its place in the current pool. In this way ReACT was able to also search the parameter space while it was solving new instances. Although straightforward in its implementation, the original work showed that ReACT could very quickly find high quality parameterizations in realtime.

This paper, extends the ReACT approach by introducing enhancements to every part of the original. Specifically, we show which parameterizations should be evaluated once a new instance arrives. We also show how and which instances should be introduced into the pool of potentials to be evaluated. Most importantly, we show how a ranking scheme commonly utilized to rank players in games, can be exploited to more accurately measure the quality of the parameterizations in the current pool. By introducing all of these changes, we show how the resulting approach ReACTR, Realtime Algorithm Configuration through Tournament Rankings, improves over the original, and even surpasses existing static configuration techniques when evaluated in a static setting.

2 Approach

Demonstrated as a powerful approach in practice, ReACT showed that it was feasible to configure a solver in realtime as new instances were being solved. At its core, the algorithm can be abstracted to Algorithm 1, taking in four parameters: number of available cores n, parameterized solver s, a stream of instances I, and the size m of the parameter pool to keep track of. Internally, a pool P of parameterizations is initialized, where each has a corresponding score recorded in E. Solving the stream of instances one at a time, a collection of n competitors C is selected to tackle the instance and all are evaluated in parallel. Here, as soon as any parameterization finishes solving the instance, all are terminated, and the results are recorded in e. Finally, the scores are processed and the pool of potential solvers is updated to replace any poorly performing parameterizations with a new set.

Originally, the pool of parameterizations was initialized by creating random assignments for all the parameters. To ensure that the performance was always improving, the current best parameterization was always among the evaluated competitors, while all others were sampled uniformly at random. This way, once one parameterization finished solving the instance it was recorded as defeating all the other competitors. An internal counter would subsequently keep track of the number of times a configuration defeated another. Using this counter, a parameterization was removed as soon as any other parameterization defeated it twice as many times as the two competed, as long as there were at least a minimum number of direct comparisons of the two. Finally, all removed configurations were replaced by new randomly generated ones.

Algorithm 1 Components of the ReACT algorithm

```
function ReACT($n$, $s$, $I$, $m$)
    $P \leftarrow$ initializeParameterizations($s, m$)
    $E \leftarrow 0^n$                         ▷ an $n$ vector initialized to 0
    for $i \in I$ do
        $C \leftarrow$ getCompetitors($P, n$)
        $e \leftarrow$ solveInstance($C, i$)
        $E \leftarrow$ updateScores($e$)
        $P, E \leftarrow$ updateParameterizations($P, E$)
```

It follows that the main reason for ReACT's success was that it always ran the best parameterization, and had a very aggressive removal policy, throwing out anything as soon as it was even hinted to be subpar. The trouble with this strategy, however, was that it failed to store a history of candidate configuration successes. Due to this, a new configuration could potentially remove a tried and tested candidate simply by getting a few lucky runs when it was first added to the pool.

Therefore, it should be clear from Algorithm 1 that the success of ReACT resides in the strategies used for each of the steps. Particularly important is the consideration of which configurations to evaluate next, which parameterizations should be discarded, and how new parameterizations should be added to the pool. This paper targets each of these questions, showing how the employment of a leaderboard of the current pool facilitates all other decisions. The remainder of this section details the strategies employed to develop ReACTR.

2.1 Ranking Parameterizations

In the world of competitive games, it is critical to have an unbiased way to compare an individual's performance to everyone else. A trivial way to do this is to simply have everyone play everyone else. Naturally, for popular games like chess, Go, checkers, etc, there are a plethora of people playing at any given time, with players coming and going from the rankings on a whim. This is also the situation that ReACT faces internally. At any given time there are a predetermined number of competitors in the pool, which can be removed and replaced by new entrants. At the same time, ReACT needs a method for quickly determining the comparative quality of each of the contestants in the pool, to know the ones worth utilizing and those that can be discarded. It therefore makes sense to employ a leader-board ranking algorithm.

Historically, in order to improve the chess rating system at the time, the Elo rating system [Elo, 1978] was introduced with the idea that the point difference in two players' ratings should correspond to a probability predictor on the outcome of a match. Specifically, and somewhat arbitrarily, the Elo rating was designed so that a difference of 200 points corresponded to an expected score of 0.75,[1] with an average player having 1500 points. After a tournament if it was found that the actual score (S_A) was higher (lower) than the expected score (E_A), it was assumed that the rank was too low (high) and thus needed to be adjusted linearly as:

$$R'_A = R_A + K(S_A - E_A)$$

where K was a constant limiting the rate of change.

An alternative version was later introduced in Glicko-2 [Glickman, 2012], which has a built-in measure of the accuracy of a player's rating, RD, as well as the expected fluctuation in the rating, σ. Specifically, the method was designed so that the more games a player is involved in, the more confident Glicko-2 was about the ranking, and the more consistently a player performs, the lower the volatility. To represent this, a Glicko-2 rating is provided as a 95% confidence range, with the lower (upper) value being the rank minus (plus) twice the RD.

While both Elo and Glicko-2 are still in use to this day, they are primarily designed for two player games. Therefore, for games involving 3+ players, all combinations of pairs must be created and updated independently in order for either of these classic approaches to work. To solve this multi-player problem for online games, the Bayesian ranking algorithm TrueSkill [Herbrich *et al.*, 2006] was invented. Like Glicko, TrueSkill measures both a player's average skill, μ, as well as the degree of uncertainty (standard deviation), σ, assuming a Gaussian distribution for a player's skill. TrueSkill uses Bayesian inference in order to rank players. After a tournament, all competitors are ranked based on the result, with the mean skill shifting based on the number of players below a player's rank and the number above, weighted by the difference in initial average ratings. The result of this is that a player that is expected to win (higher μ value) gains little by beating a lower ranked opponent. However, if a lower ranked player beats a higher ranked player, then the lower ranked player will receive a large increase in their μ value. As a player competes in more tournaments TrueSkill becomes more confident in the μ that is assigned and so the uncertainty value, σ is reduced after every tournament played.

For ReACTR, the confidence metric provided by Glicko-2 and TrueSkill is a highly desirable feature that could help determine whether it is worth continuing to evaluate a parameterization or whether it can be safely discarded. We implemented and experimented with both ranking algorithms, with Figure 1 showing a typical result. What the figure shows is the cumulative time the ReACTR algorithm with the two ranking methods needs to go through all instances from a particular benchmark. While we will describe the benchmarks later in the paper, what is important to note here, is that in a typical situation, TrueSkill is able to help bring better parameterizations to the top, resulting in better overall performance in the long run. For this reason that in all subsequent parts of this paper we only rely on the TrueSkill ranking methodology. We use a highly-rated open source Python implementation of TrueSkill for our experiments [Zongker, 2014] using all of the default settings (μ = 25, σ=8.3').

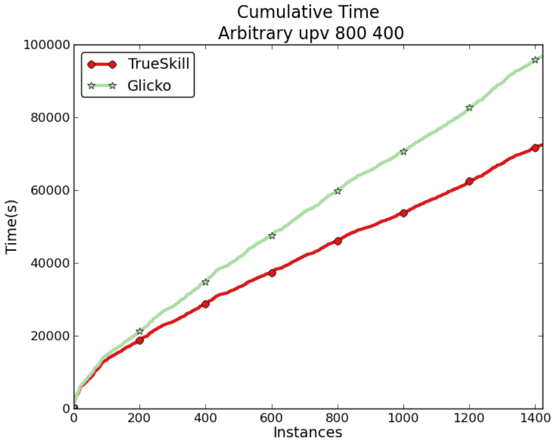


Figure 1: Cumulative solving time using Glicko and TrueSkill for ranking.

2.2 Choosing the Competitors

Using TrueSkill for ranking means that each member in our current pool of thirty potential parameterizations has an associated score declaring its quality, as well as confidence rating of this score. To guarantee that the overall solver will never produce bad performance, the best known configuration is always among those that is evaluated. But other than that, it is not immediately clear exactly which other parameterizations should be evaluated in each tournament. We refer to the method used to select the configurations to run from the leader-board as the sampling strategy.

Restricting ourselves to only selecting six configurations, the number of cores we typically use for parallel execution, we compare several strategies on an assortment of auction problems. Again, these problems and the parameterized solver we use are described in detail in a later section. For now, we treat this dataset and the thirty randomly generated parameterizations as a black box. The objective is to choose to run solvers such that the cumulative time necessary to go through all the instances is minimized.

Utilizing this setup, and the TrueSkill approach to rank the solvers in our pool we compare five sampling strategies. Specifically we compare strategies that take the top n solvers and the $6 - n$ solvers uniformly at random. The results are shown in Figure 2.

What we observe in this figure is that running the top three and a random set of three is the worst strategy, resulting the solver spending the most time needing to go through all the instances. Alternatively, running just the top single solver and five random ones allows ReACTR to better find a good parameterization in the pool, and thus lowering the total time. Having said this, we also note that there is not much difference in which strategy is selected. Therefore, we hedge our

[1] Expected score is the probability of winning plus half the probability of a tie.

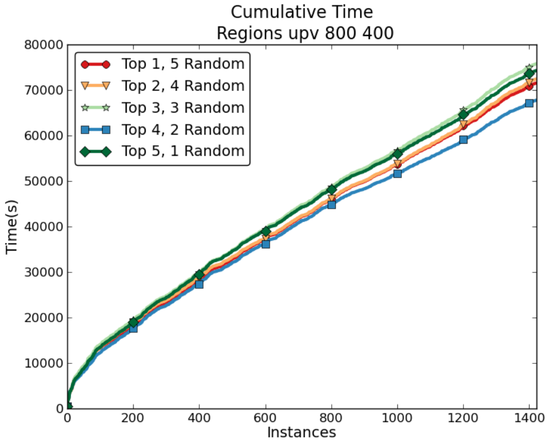


Figure 2: Cumulative solving time for ReACTR with different sampling strategies.

bets by, in practice, running the top two known solvers and the others chosen informally at random. We intuitively assume that by running the top two solvers we would be in better shape in knowing the quality of the best solvers.

2.3 Cleaning the Pool

Even with a high quality ranker, such as TrueSkill, if there are no good parameterizations in our pool of contenders, we will never be able to improve our overall performance. Therefore it is imperative for ReACTR to constantly discard the obviously inferior parameterizations and to replace them with new contenders. This leaves the question as to which strategy to adopt for removing parameterizations. A more aggressive removal strategy allows us to evaluate more new parameters thus increasing the chances of finding better parameterizations, however, this comes with more risk of removing good parameterizations prematurely. A more conservative removal strategy allows us to evaluate each parameterization more fully but because parameterizations are removed more slowly less of the configuration space is explored. The questions we must answer are therefore how many configurations should we keep and how confident should we be that a configuration is poor before removing it.

Because the evaluation of ReACTR on real data requires us to run a solver for non-trivial amount of time over a large number of instances, finding the best strategy for removing instances can be extremely expensive. To overcome this, we simulate the process using synthetic data. Specifically, we know that because our instances are relatively homogeneous in practice, any parameterization of a solver would have a particular expected performance with some variance. We further assume that there is a particular mean expected performance and a mean variance. Therefore, each parameterization of a solver is simulated by a normal distribution random number generator with a fixed mean and standard deviation. Once this simulated solver is removed, it is replaced by another one, where the new mean and variance are assigned randomly according to a global normal random number generator. What this means is that most of our simulated solvers have a similar performance, with those being significantly better than others being increasingly unlikely. Because in practice we introduce new parameterizations sampled uniformly at random, this methodology of representing solvers is very close to reality, as independent (not shown) tests revealed.

Given these simulated solvers, the objective of ReACTR is to find a solver that leads to the shortest cumulative solving time for 500 instances. Of course since these solvers are random we repeat the experiment several times. What we observe is displayed in Figure 3. Specifically we observe that the best strategy for removing parameterizations from the pool of contenders is by keeping the top 15-20 solvers and anything in which we have a greater than 5.0-5.5 uncertainty rating.

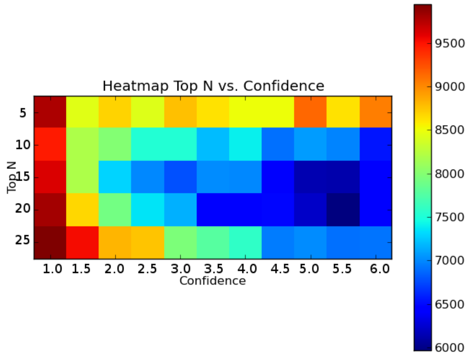


Figure 3: Heat-map showing the combined effect of the number of kept parameterizations and TrueSkill's confidence rating(lower values indicate higher confidence). The colorbar shows the cumulative solving time in seconds.

2.4 Replenishing the Pool

Every parameterization which is removed from the leaderboard must be replaced by a newly generated configuration. In order to balance exploration of new configurations and the exploitation of the knowledge we have already gained, ReACTR uses two different generation strategies. Diversity is ensured by generating parameterizations where the value of each parameter is set to a random value from the range of allowed values for that parameter.

Additionally, ReACTR exploits the knowledge it has already gained through ranking by using a crossover operation similar to that used in genetic algorithms. For this, two parents are chosen from among the five highest ranked configurations then with equal probability each parameter takes one of the parents values. Additionally, like a standard genetic algorithm, some small percentage of the parameters are allowed to mutate. That is, rather than assuming one of the parent values, a random valid value is assigned instead.

We allow for a variable to control the balance of exploitation to exploration, or the percentage of generated vs random parameterizations we introduce. Again for now treating the specific collection of instances and parameterized solver as a blackbox, Figure 4 shows the effect on cumulative time when ReACTR varies the ratio of the amount of exploitation to do.

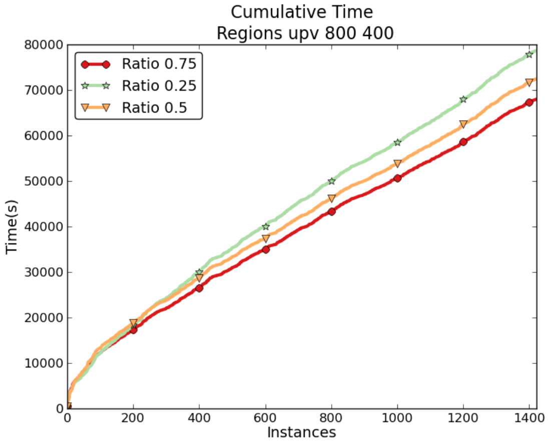


Figure 4: Cumulative solving time for ReACTR with different exploitation ratio settings.

From this graph, we can see that, at least in the relation of these two methodologies, it is much better to actively exploit existing knowledge, generating a greater proportion of parameterizations using crossover.

3 Experimental Setup

We evaluate the ReACTR methodology on three datasets. The first two are variations of combinatorial auction problems which were used in the evaluation of the original ReACT configurator. A combinatorial auction is a type of auction where bids are placed on groups of goods rather than single items. These instances are encoded as mixed-integer programming (MIP) problems and solved using the state-of-the-art commercial optimizer IBM CPLEX [IBM, 2014]. The problems were generated using the Combinatorial Auction Test Suite (CATS) [Leyton-Brown *et al.*, 2000], which can generate instances based on five different distributions aiming to match real-world domains.

The two combinatorial auction datasets are generated based on the regions and arbitrary domains. Regions simulates combinatorial auctions where the adjacency in space of goods matters. In practice, this would be akin to selling parcels of land or radio spectrum. Alternatively, the arbitrary domain foregoes a clear connection between the goods being auctioned, as may be the case with artwork or antiques. These two domains generate instances which are dissimilar enough to warrant different strategies for solving them and by extension different parameter configurations for the solver.

The regions dataset was generated with the number of goods set to 250 (standard deviation 100) and the bid count to 2,000 (standard deviation 2,000). Similarly, 800 goods (standard deviation 400) and 400 bids (standard deviation 200) were used for the arbitrary dataset. These particular values were chosen for generation as they produce diverse instances that are neither too easy nor too hard for our solver. Furthermore the dataset is cleaned by removing any instance which is solvable in under 30 seconds using the CPLEX default settings. These are removed because they are quickly solvable even by poor configurations and in practice usually handled by a pre-solver. Similarly, instances that take more than 900 seconds to solve using the CPLEX defaults were removed as these are considered too hard and little information is gained where all solvers time-out. After this filtering the regions dataset contains 2,000 instances (split into a training set of 200 and a test set of 1800) while the arbitrary dataset has 1,422 instances (200 training and 1,222 test). The solver time-out for both datasets is set to 500 seconds.

The third dataset comes from the Configurable SAT Solver Competition (CSSC) 2013 [UBC, 2013]. This dataset was independently generated using FuzzSAT [Brummayer *et al.*, 2010]. FuzzSAT first generates a boolean circuit and then converts this to CNF. We solve these instances using the popular SAT solver Lingeling [Biere, 2010]. The circuits were generated using the options `-i 100` and `-I 100`. Similar to what was done with the auction datasets, any instances that could be solved in under 1 second were removed. The resulting dataset contained 884 instances which was split into 299 training and 585 test. We use a timeout of 300 seconds for this dataset, the same as that used in the CSSC 2013.

It is important to note here, that ReACTR by its nature is an online algorithm, and therefore does not require a separate training dataset. However, in order to compare to existing methodologies, a training set is necessary for those approaches to work with.

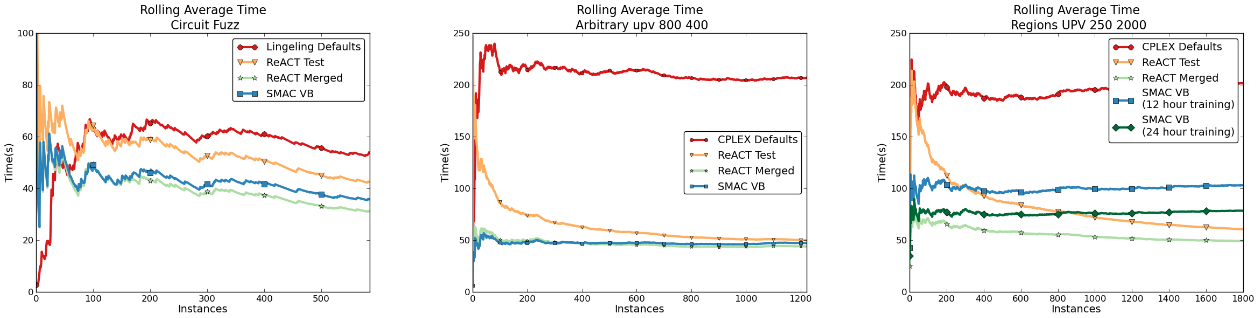
All experiments were run on a system with 2 X Intel Xeon E5430 processors(2.66Ghz) and 12 GB RAM. Though there are 8 cores available on each machine we limit ourselves to 6 so as not to run out of memory or otherwise influence timings.

4 Results

We show two separate scenarios for ReACTR. First, we consider a scenario where there is a collection of training data available beforehand, or alternatively a training period is allowed. We refer to this approach as “ReACTR Merged”, where the configurator is allowed to make a single pass over the training instances to warm-start its pool of configurations. Secondly, we evaluate “ReACTR Test”, which assumes no prior knowledge of the problem, and starts the configuration process only when it observes the first test instance.

For comparison we evaluate both versions of ReACTR against a state-of-the-art static configurator, SMAC. For completeness, we investigated SMAC when trained for 12, 24, 36 and 48 hours. This way we cover scenarios when a new solver is configured each night, as well as the best configuration SMAC can find in general. We observed that on our particular datasets performance improvements stagnated after 12 hours training, except in the case of the regions dataset (for which we show the 24 hour training). Furthermore, because ReACTR uses six cores, six versions of SMAC are trained using the ’shared model mode’ option, which allows multiple SMAC runs to share information. Upon evaluation, all six configurations are run and the time of the fastest performing SMAC tuning on each instance is logged. By doing this the CPU time used by SMAC and ReACTR is comparable.

We also show the results for both the previous version of ReACT (on the merged dataset described above), “ReACT



(a) Circuit fuzz using Lingeling.

(b) Arbitrary CA dataset using CPLEX.

(c) Regions CA using CPLEX.

Figure 5: Rolling average solving time on various dataset and solver combinations.

Table 1: Summary of training, testing and total time needed for the various configurations on the benchmark datasets.

		Time taken (1000s)						
		Solver	ReACTR		SMAC (number of hours)			
		Default	Test	Merged	12	24	36	48
Regions	Train	0	0	24	43	86	130	173
	Solve	363	109	**88**	185	141	110	102
	Total	363	**109**	112	228	227	240	275
Arbitrary	Train	0	0	19	43	86	130	173
	Solve	253	61	**54**	58	57	57	57
	Total	253	**61**	72	101	143	186	230
Circuit Fuzz	Train	0	0	13	43	86	130	173
	Solve	32	25	**18**	21	21	21	21
	Total	32	**25**	31	64	107	150	194

Merged" and the default solver parameterizations.

Figure 5(a) shows the rolling average (total time to date/instances processed) on the circuit fuzz dataset. We can see that both versions of ReACTR easily outperform the Lingeling defaults. What is more interesting is that ReACTR is able to outperform SMAC (trained for 12 hours) after a single pass over the training set (taking under 4 hours). Even without the warm-start, ReACTR is able to find parameters that significantly better than the defaults and not too far off those that were otherwise configured.

In Figure 5(b), we see that on the Arbitrary Combinatorial Auction dataset, configuration is extremely important, and that all configurators are able to find the good parameterizations. In Figure 5(c), however, we once again see that both versions of ReACTR find significantly better parameterizations than those that can be found after 12 hours of tuning SMAC, and even the configuration found after 24 hours of tuning SMAC.

Finally, Table 1 shows the amount of time each configuration technique requires to go through the entire process. This shows the amount of time needed to train the algorithm and the amount of time needed to go through each of the test instances. The times are presented in 1000s of seconds. Note that in all cases, ReACTR Merged requires less time to train and also finds better configurations than SMAC. However, if training time is a concern, then ReACTR Test requires significantly less total time than any other approach.

5 Conclusion

It is clear from multitudes of past examples that whenever one needs to use a solver on a collection of instances, it is imperative to utilize an algorithm configurator to automatically set the solver's parameters. But while there are now a number of existing high quality configurators readily available, unfortunately all assume a static view of the world. In practice, training instances are not always available beforehand, problems tend to change over time, and there are times when there is no extra time to train an algorithm offline. It is under these cases, that ReACT was shown to thrive, able to achieve high quality parameterizations while tuning the solver as it was solving new instances. This paper, showed how the crucial components of the original ReACT methodology could be enhanced by incorporating the leader-board ranking technique of TrueSkill. The resulting method, Realtime Algorithm Configuration through Tournament Rankings (ReACTR), was shown to surpass even a state-of-the-art configurator SMAC across multiple domains.

Acknowledgments

This publication has emanated from research conducted with the financial support of Science Foundation Ireland (SFI) under Grant Number SFI/12/RC/2289.

References

[Amadini *et al.*, 2014] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. Sunny: a lazy portfolio approach for constraint solving. *Theory and Practice of Logic Programming (TPLP)*, pages 509–524, 2014.

[Ansótegui *et al.*, 2009] Carlos Ansótegui, Meinolf Sellmann, and Kevin Tierney. A gender-based genetic algorithm for the automatic configuration of algorithms. In *Principles and Practice of Constraint Programming-CP 2009*, pages 142–157. Springer, 2009.

[Biere, 2010] Armin Biere. Lingeling. *SAT Race*, 2010.

[Brummayer *et al.*, 2010] Robert Brummayer, Florian Lonsing, and Armin Biere. Automated testing and debugging of sat and qbf solvers. In *Theory and Applications of Satisfiability Testing–SAT 2010*, pages 44–57. Springer, 2010.

[Elo, 1978] Arpad E Elo. *The rating of chessplayers, past and present*, volume 3. Batsford London, 1978.

[Fitzgerald *et al.*, 2014] Tadhg Fitzgerald, Yuri Malitsky, Barry O'Sullivan, and Kevin Tierney. React: Real-time algorithm configuration through tournaments. In *Proceedings of the Seventh Annual Symposium on Combinatorial Search, SOCS 2014, Prague, Czech Republic, 15-17 August 2014.*, 2014.

[Glickman, 2012] Mark E Glickman. Example of the glicko-2 system. *Boston University*, 2012.

[Herbrich *et al.*, 2006] Ralf Herbrich, Tom Minka, and Thore Graepel. Trueskill: A bayesian skill rating system. In *Advances in Neural Information Processing Systems*, pages 569–576, 2006.

[Hutter *et al.*, 2009] Frank Hutter, Holger H Hoos, Kevin Leyton-Brown, and Thomas Stützle. Paramils: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36(1):267–306, 2009.

[Hutter *et al.*, 2011] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Learning and Intelligent Optimization*, pages 507–523. Springer, 2011.

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

[Leyton-Brown *et al.*, 2000] Kevin Leyton-Brown, Mark Pearson, and Yoav Shoham. Towards a universal test suite for combinatorial auction algorithms. In *Proceedings of the 2nd ACM conference on Electronic commerce*, pages 66–76. ACM, 2000.

[Malitsky and Sellmann, 2012] Yuri Malitsky and Meinolf Sellmann. 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*, pages 244–259. Springer, 2012.

[Malitsky *et al.*, 2013a] Yuri Malitsky, Deepak Mehta, and Barry O'Sullivan. Evolving instance specific algorithm configuration. In *Sixth Annual Symposium on Combinatorial Search*, 2013.

[Malitsky *et al.*, 2013b] Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. Algorithm portfolios based on cost-sensitive hierarchical clustering. In *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, pages 608–614. AAAI Press, 2013.

[UBC, 2013] UBC, 2013. Configurable SAT Solver Competition.

[Xu *et al.*, 2008] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Satzilla: portfolio-based algorithm selection for sat. *Journal of Artificial Intelligence Research*, pages 565–606, 2008.

[Zongker, 2014] Doug Zongker. trueskill.py, 2014. https://github.com/dougz/trueskill.