Automatically Configuring Algorithms for Scaling Performance

James Styles[1], Holger Hoos[1], and Martin Müller[2]

[1] University of British Columbia, 2366 Main Mall, Vancouver, BC, V6T 1Z4, Canada
{jastyles,hoos}@cs.ubc.ca
[2] Computing Science, University of Alberta, Edmonton, AB, T6G 2E8, Canada
mmueller@ualberta.ca

Abstract. Automated algorithm configurators have been shown to be very effective for finding good configurations of high performance algorithms for a broad range of computationally hard problems. As we show in this work, the standard protocol for using these configurators is not always effective. We propose a simple and computationally inexpensive modification to this protocol and apply it to state-of-the-art solvers for two prominent problems, TSP and computer Go playing, where the standard protocol is unable or unlikely to yield performance improvements, and one problem, mixed integer programming, where the standard protocol is known to be effective. We show that our new protocol is able to find configurations between 5% and 75% better than the standard protocol within the same time budget.

1 Introduction

Many high performance algorithms for computationally hard problems have numerous parameters, some exposed to end users and others hidden as hard-coded design choices and magic constants, that control their behaviour and performance. Recent work on automated configurators has proven to be very effective at finding good values for these parameters [4, 6, 7, 8, 12, 13, 14, 16, 17]. The standard protocol for using automated configurators, such as ParamILS [14], to optimize the performance of a parametric algorithm for a given problem is as follows:

1. Identify the intended use case of the algorithm (*e.g.*, structure and size of expected problem instances, resource limitations) and define a metric to be optimized (*e.g.*, runtime).
2. Construct a training set/scenario which is representative of the intended use case. The performance of the configurator depends on being able to evaluate a large number, ideally thousands, of configurations. Training instances/scenarios must be chosen to permit this.
3. Perform multiple independent runs of the configurator, typically 10-25 [14].
4. Validate the final configurations found by each run on the training set.
5. Select from the final configurations found by the independent runs the one with the best performance on the training set.

While this protocol has been successfully applied to many problems and solvers, we have observed that it is not always feasible. In particular, for Step 2, choosing a training set becomes problematic if the time taken to evaluate a configuration on the training settings is too large. This is the case for Fuego [9], a state-of-the-art computer Go player based on Monte Carlo tree search (MCTS). The time taken to evaluate a single configuration of Fuego for competition level play can take hours or even days (see Section 5.3). For ParamILS to have any hope of finding a good configuration, each run would have to be allowed to take several years. In situations of this nature, a training set significantly easier than the intended use case must be used. Unfortunately, the use of easier training sets may lead to configurations whose performance may not scale up to the intended use case.

In this paper, we explore a simple modification to the standard protocol for using automated configurators that attempts to resolve this problem in a generic manner. We apply our new protocol to three well-known problems and configuration scenarios. The first of these is the traveling salesperson problem (TSP), a widely studied combinatorial optimization problem with numerous industrial applications, for which we configure Keld Helsgaun's implementation of the Lin-Kerninghan algorithm (LKH) [11], the best incomplete solver for TSP currently known (and far superior to any complete solver in terms of finding optimal or near-optimal solutions fast). In an early stage of our work, described in Section 2, we found that the standard protocol is ineffective for this configuration scenario. Our second scenario concerns computer Go playing, a grand challenge in artificial intelligence, using the state-of-the-art Monte Carlo tree search (MCTS) based player Fuego [9]. Evaluating configurations for this scenario requires playing hundreds of games (see Section 4.3) which becomes prohibitively expensive for the intended use case. This makes using the standard protocol infeasible. The third scenario we consider involves solving mixed integer programming (MIP) problems, which are widely used for representing constrained optimization problems in academia and industry, using the state-of-the-art commercial solver CPLEX [2]. Unlike for the other two scenarios, the standard protocol has been proven to be very effective for this configuration scenario [12]; our primary motivation for studying it here is to verify that our new protocol does not lead to compromised configuration performance in cases where the standard protocol is already effective.

The remainder of this paper is structured as follows. Section 2 illustrates the problems we have encountered using the standard protocol for configuring LKH. Section 3 presents our new protocol. Section 4 describes the three configuration scenarios we consider in this work in more detail. Section 5 explains the experimental setup we used for evaluating our new configuration protocol, and Section 6 presents the empirical results we obtained. Section 7 provides conclusions and an overview of ongoing and future work.

2 A first attempt at configuring LKH

The starting point for this work was an attempt to configure LKH [11] using ParamILS. In particular, we were interested in reducing the time taken to find

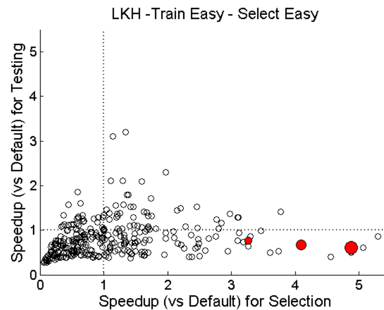

Fig. 1. Comparison between the selection and testing (PAR10) speedup, relative to the default configurations, for 300 configurations of LKH found by ParamILS using the easy instances, see Section 4.1, for both training and selection. Configurations shown in the CPU time column of Table 1 are filled in and coloured red. The size of the points for these configurations corresponds to the time required for finding them.

optimal (or near optimal) solutions for structured instances like those found in the well-known TSPLIB benchmark collection [15], with a focus on instances containing several thousand cities, on which LKH's default configuration can take several CPU hours on our reference machines (see Section 5) to find near-optimal solutions. As ParamILS requires thousands of evaluations [14] of an algorithm to reliably achieve good results, using such instances directly for training would result in individual configuration experiments with a duration of up to one year. Since this is infeasible, we decided to perform training on similarly structured, but significantly smaller (200–1500 node) instances which takes less than a CPU minute for the default configuration of LKH to solve.

Since LKH is an incomplete solver, there is no guarantee on the quality of solution found by a given run. We are therefore interested in the runtime of a configuration on an particular instance as well as the quality of solution found. To avoid constructing a Pareto front, we combine these two raw performance metrics using penalized average runtime (PAR10). This metric uses the total running time of a given run and then penalizes runs which are unable to achieve a target solution quality within some time cutoff. As we do not know the optimal solution quality for every instance used, we determine the target solution quality by using the final solution found by a single long run of the default configuration. On instances with a known optimal solution, the target quality chosen is often equivalent to the optimal for small instances and within 1% of the optimal for large instances.

Following the standard protocol, we performed multiple independent 24 CPU-hour runs of ParamILS using this easier training set and optimizing for penalized average runtime (PAR10). The configurations found by these experiments performed very well on the training set, but often turned out to be worse than the default configuration when evaluated on the testing set, consisting of the larger instances we were ultimately interested in solving.

To further explore the reasons for this apparent failure of the standard protocol, we expanded our experiment to include 300 independent 24-hour runs of ParamILS using the same metric and the same training set and evaluated the final configuration found by each of these runs on the entire testing set. As seen in Figure 1, we found that while the standard protocol for ParamILS was able to find good configurations, it was unlikely to select them. This is due to the fact that the performance of a configuration of LKH on the training set is not a good predictor of that configuration's performance on the testing set. Despite being ineffective as a predictor of testing set performance, the training set was able to guide runs of ParamILS to configurations with a speedup factor of up to 3.19, which suggests it still has value in the configuration process.

3 Automated algorithm configuration for scalable performance

To address the problem encountered in Section 2, we devised a simple modification to the standard protocol for using configurators such as ParamILS. Instead of selecting between the final configurations found in independent configurator runs based on their performance on the training set, we select based on their performance on a set of intermediate instances that are harder than the training set, but easier than the testing set. For this work, we define intermediate difficulty based on percentiles of the distribution of running time for the default configuration of a given solver over the testing set. This protocol has three advantages over alternative approaches: (1) it does not require any modifications to the underlying configurator; (2) it can reuse the results of existing configuration experiments; and (3) it can be set up to require only a moderate amount of additional processing time (in our experiments, the overhead is always below 50% of the total time budget). To assess this protocol, which we dubbed *Train-Easy, Select-Intermediate (TE-SI)*, we compare it to the original protocol, *Train-Easy, Select-Easy (TE-SE)*, and to an alternative approach, in which training is directly performed on the harder instances used for selection, *Train-Intermediate, Select-Intermediate (TI-SI)*, always correctly accounting for the overhead required for evaluating configurations at the selection stage.

4 Configuration scenarios

4.1 Solving TSP using LKH

LKH [11] is a two-phase, incomplete solver for the TSP. It first performs deterministic preprocessing using subgradient optimization, which modifies the cost function of the given TSP instance while preserving the total ordering of solutions by tour length. The main goal of this first phase, which can sometimes already reach the desired solution quality, is to make an instance easier for the subsequent phase to solve. The second phase consists of a stochastic local search procedure based on chaining together so-called k-opt moves.

For the following experiments, we used a version of LKH 2.02, which we have extended to allow several parameters to scale with instance size and to make

use of a simple dynamic restart mechanism to prevent stagnation behaviour we had observed in preliminary experiments. The original configuration space is preserved by these modifications (*i.e.*, it is possible to replicate the behaviour of any configuration for the original LKH 2.02 using our extended version).

Training and testing were done using instances from the well-known TSPLIB benchmark collection [15]. TSPLIB is a heterogeneous set consisting mostly of industrial and geographic instances. The original TSPLIB set contains only 111 instances; since we consider this too small to allow for effective automated configuration and evaluation, we generated new TSP instances based on existing TSPLIB instances by randomly selecting 10%, 20%, or 30% of the existing instance's nodes to be removed. These TSPLIB-like instances retain most of the original structure and are comparable in difficulty to the original instance, ranging from requiring a factor of 30 less time to a factor of 900 more time for the default configuration of LKH to solve.

The modified version of LKH 2.02 and the TSPLIB-like instances will be made available on our website upon publication.

4.2 Solving MIP using CPLEX

CPLEX is one of the best-performing and most widely used solvers for mixed integer programming problems. It is based on a highly parameterized branch-and-cut procedure that generates and solves a large number of linear programming (LP) subproblems. While most details of this procedure are proprietary, at least 76 parameters which control CPLEX's performance while solving MIP problems are exposed to end users.

Our work on this scenario aims to mirror recent work by Hutter *et al.* [12] for configuring CPLEX 12.1 on the CORLAT instance set, for which the standard protocol for using ParamILS was able to achieve a 52-fold speedup over the CPLEX default settings. The CORLAT instance set consists of 2000 instances based on real data modeling wildlife corridors for grizzly bears in the Northern Rockies [10]. Our goal in considering this scenario is to show that our new configuration protocol is effective even in scenarios where the default protocol is known to work well.

Hutter *et al.* [12] used CPLEX 12.1, the most recent version available at the time of their study. CPLEX 12.3, the current version at the time of this writing, performs significantly better on the CORLAT instances, achieving a speedup factor of up to 90 on the hardest instances in the set. To compensate for this significant improvement of the default configuration, we performed a 1/50th time scale replica of the configuration experiments conducted by Hutter *et al.* [12]. Reducing the runtime of ParamILS and the per-instance cutoffs preserves both the percentage of training instances that the default configuration is capable of solving within the time cutoff as well as the number of evaluations ParamILS is able to perform within the total configuration time budget. The results of our experiments depend only on the ratio of per-instance runtime to total configuration time and are invariant with respect to the overall time scale. While we used significantly reduced configuration times, we believe that our results should generalize to longer configuration runs using harder instances.

The metric being optimized for this scenario is penalized average runtime (PAR10). We measure the total running time of given run and penalize if it is unable to find the optimal solution within a cutoff of 1 hour for testing, 6 seconds for training on easy and 24 seconds for training on intermediate. CPLEX is a complete solver, so every run of a configuration that does not exhibit errant behaviour is guaranteed to find the optimal solution to every instance given enough computational resources. For this scenario, cutoffs and penalties are only used to limit the total computational effort of performing these experiments.

We also applied the TE-SI protocol to CPLEX 12.1. We found configurations that offered significant speedups (≥ 50) compared to the default configuration of 12.1, improving upon the results found in [12]; however, the overall result remained qualitatively similar to our work with CPLEX 12.3. We do not present our results for CPLEX 12.1 due to space limitations.

4.3 Playing Go using Fuego

Developing programs for the game of Go has been a topic of intense study over the last five decades. Only recently, with the advent of Monte Carlo Tree Search (MCTS), has the strength of Go programs caught up with top human players, at least on small board sizes (up to 9×9). MCTS combines position evaluation by randomized *playouts* of the remainder of a game with a new, selective search approach that balances exploration and exploitation: the algorithm combines exploration of parts of a game tree that are still underdeveloped with exploitation by deep search of the most promising lines of play. The open-source project Fuego [9] contains both a game-independent framework for MCTS and a state-of-the-art Go program. The program was the first to beat a top human professional player in an even game on the 9×9 board and has won numerous computer competitions [1].

Like the other configuration scenarios we study in this work, Fuego has a large number of configurable parameters. The performance metric to be optimized is the win rate of a configuration when played against the default configuration. Note that the baseline is not necessarily 50%: For certain board sizes and playout limits, the default configuration is stronger playing black than white, while for other board sizes and playout limits, the opposite holds.

Noisy evaluations Since Fuego uses a randomized playout strategy in its core MCTS procedure, the win rate of any set of test games played with Fuego varies. This introduces a significant source of noise when evaluating configurations of Fuego. This noise must be compensated for by playing additional games; otherwise, the observed win rates are meaningless (*e.g.*, with 10 games played, there is a more than 40% chance that the observed win rate of a configurations differes from its true win rate by at least 10%). The exact number of games needed depends on the true win rates of the configurations being compared, but often hundreds, if not thousands, of games are required to reduce the chance of incorrectly ranking two configurations to less than 1%. A key point is that the closer two configurations are in true win rate, the more games are needed to correctly rank them. We note that while in principle, similar concerns arise for many

other configuration scenarios involving randomised algorithms, the amount of evaluation noise in the case of Fuego (and other randomised game players) is particularly large, due to the fact that individual games have binary outcome.

This is particularly problematic for automatic configurators like ParamILS, which often rely on a sequence of small incremental improvements to a configuration. If the improvement is too small, then it will be dwarfed by the noise in the evaluations and it is impractical to play a sufficient number of games, potentially thousands, to adequately compensate. We compromise by playing as many games as are necessary to evaluate a configuration, up to a limit of 200, during training. We note that when comparing two configurations with true win rates (as determined in playing against some reference configuration) within 1% of each other, there is a 20.7% chance of incorrectly ranking them based on a set of 200 games.

5 Experimental setup

For each configuration scenario, we defined three instance sets of distinct difficulties: an easy instance set, designed to allow ParamILS to perform at least several hundred evaluations of candidate configurations; a hard instance set, designed to represent the difficulty of instances/situations that we are interested in optimizing the target algorithms performance for; and a set of intermediate instances with a difficulty between the easy and hard instances. The exact definition of easy, intermediate and hard is specific to each the configuration scenario.

Using these sets, we performed three sets of configuration experiments using independent runs of ParamILS. (We chose ParamILS, because it is the only readily available algorithm configuration procedure that has been demonstrated to work well on configuration scenarios of the difficulty considered here.) In the first set of experiments, we used the easy instances during training and then selected a configuration, from the set of the final configurations produced across a number of independent runs of ParamILS, according to its performance on the same (easy) set (this is the standard protocol, TE-SE). The second set used the easy instances set for training, but intermediate instances for selection (this is our new protocol, TE-SI). The third set used the intermediate instances for both training and selection (TI-SI). All testing was performed on the hard instances. (Recall that we are interested in the case where the hard instances are too difficult to be used in training.)

LKH and CPLEX experiments were performed on the 384 node DDR partition of the Westgrid Orcinus cluster; Orcinus runs 64-bit Red Hat Enterprise Linux Server 5.3, and each node has two quad-core Intel Xeon E5450 64-bit processors running at 3.0 GHz with 16GB of RAM.

Fuego experiments were performed on the 512 node Westgrid Lattice cluster. Lattice runs 64-bit Linux CentOS 5.5, and each node has two quad-core Intel Xeon L5520 64-bit processors running at 2.27 GHz with 12 GB of RAM.

5.1 Solving TSP using LKH

The hard instance set consists of 3192 instances containing up to 6000 cities, drawn from both the original TSPLIB and TSPLIB-like instances. The default configuration of LKH takes approximately 214 CPU hours on our reference machines to run on the entire set. The 99th percentile difficulty is 2900 CPU seconds.

The intermediate instance set consists of instances which take the default configuration between 350 and 580 CPU seconds to solve; this range corresponds to between 12.5 and 20 percentile difficulty found in the hard instance set. The default configuration takes approximately 20 CPU hours to run on the entire intermediate instance set. All instances in the intermediate set are drawn from a set of TSPLIB-like instances disjoint from the hard instance set. When used for training, a per-instance cutoff of 780 CPU seconds is used.

The easy instance set consists of instances which take the default configuration between 1 and 52 CPU seconds to solve. The default configuration takes 19 minutes to run on the entire easy instance set. All instances in the easy set are drawn from a set of TSPLIB-like instances disjoint from those used in the hard and intermediate sets. When used for training, a per-instance cutoff of 120 seconds is used.

Using these sets, we performed two sets of configuration experiments. The first set consists of 300 independent 24-hour runs of ParamILS using the easy set for training. The second set consists of 100 independent 24-hour runs of ParamILS using the intermediate set for training.

The TE-SE protocol requires 1459 CPU minutes per run of ParamILS. The TE-SI and TI-SI protocols require 2640 minutes per run of ParamILS.

5.2 Solving MIP using CPLEX

The hard instance set consists of 1650 instances drawn from the set of CORLAT instances used in [12]. The default configurations takes approximately 11.5 CPU hours to evaluated the entire instance set. The 99th percentile difficulty is 448 CPU seconds.

The intermediate instance set consists of instances which take the default configuration between 54 and 90 seconds to evaluate; this range corresponds to between 12.5 and 20 percentile difficulty found in the hard instance set. The default configuration takes approximately 1.1 CPU hours to evaluate the entire intermediate instance set. The instances in the intermediate instance are disjoint from the hard instance set. When used for training, a per-instance cutoff of 24 CPU seconds is enforced.

The easy instance set consists of instances which take the default configuration between 1 and 10 seconds to evaluate. The default configuration takes approximately 18 CPU minutes to evaluate the entire easy instance set. The easy instance set is disjoint from both the hard and intermediate instance sets. When used for training, a per-instance cutoff of 6 CPU seconds is enforced.

Using these sets, we performed two sets of configuration experiments. The first set consists of 100 independent 24-hour runs of ParamILS using the easy set for

training. The second set consists of 100 independent 24-hour runs of ParamILS using the intermediate set for training.

The TE-SE protocol requires 4531 CPU seconds per run of ParamILS. The TE-SI and TI-SI protocols require 7456 seconds per run of ParamILS.

5.3 Playing Go using Fuego

The hard setting consists of playing 5000 games on a 7×7 board with 300 000 playouts. For 5000 games there is a [100%, 99.8%, 91%] chance of correctly determining the true win rate of a configuration to within [3%, 2%, 1%].

The intermediate setting consists of playing 1000 games on a 7×7 board with 100 000 playouts for selection and 5000 such games for testing. For 1000 games there is a [97%, 88%, 60%] chance of correctly determining the true win rate of a configuration to within [3%, 2%, 1%].

The easy setting consists of playing 1000 games on a 7×7 board with 10 000 playouts for selection and 5000 such games for testing,

Using these sets, we performed two sets of configuration experiments. The first set consists of 80 independent 24-hour runs of ParamILS using the easy set for training. The second set consists of 80 independent 24-hour runs of ParamILS using the intermediate set for training. Each set of configuration experiments is split evenly across configuring for playing black or playing white.

The TE-SE protocol requires 5904 CPU hour per run of ParamILS. The TE-SI and TI-SI protocols require 7200 hours per run of ParamILS.

6 Results

We are interested in how effective the TE-SI protocol is in a typical setting where 10–25 independent runs of the configuration procedure are performed (tyically in parallel). To assess the variation in the results of such configuration experiments, we have performed a significantly higher number of configurator runs for each of our configuration scenarios and then performed a bootstrap analysis based on the data thus obtained.

For a specific protocol and a target number n of ParamILS runs, we generated 100 000 bootstrap samples by selecting, with replacement, the configurations obtained from the n runs. For each such sample R, we chose a configuration according to the selection criteria of the protocol under investigation and used the performance of that configuration on the testing set as the result of R.

We present the results from these analyses in two ways. In Table 1, we show the median performance of the bootstrap samples for each protocol when using different numbers of independent ParamILS runs and overall CPU time budget. In Figure 2, we show the median performance and the ranged spanned by the 5th and 95th percentile performance of bootstrap samples versus total CPU time budget. For reference, we also show the quality of the default and of the best known configuration for each scenario. The data in Table 1 thus represents several time slices from Figure 2.

Table 1. Overview of the speedup versus the default, for LKH and CPLEX, and the win rate versus the default, for Fuego, during testing for the configurations found by the three protocols. The performance of each configuration on the instances/settings used for selection is shown in parentheses. Values presented are the medians over 100 000 bootstrapped samples. The best known performance on easy, intermediate and testing instance sets / settings are provided for reference. Configurations shown in the CPU Time column are highlighted in the scatter plots in Figures 1, 3 and 4.

	Speedup Factor (PAR10) vs Default Configuration					
LKH	best easy: 5.29, best intermediate: 2.58, best testing: 3.19					
	Runs of ParamILS			CPU Time		
	10	25	50	20 Days	50 Days	100 Days
train easy select easy	0.82 (2.78)	0.73 (3.47)	0.67 (4.09)	0.76 (3.26)	0.67 (4.09)	0.61 (4.88)
train easy select inter.	**1.29** (1.29)	**1.52** (1.59)	**1.71** (1.84)	**1.29** (1.31)	**1.52** (1.61)	**1.71** (1.98)
train inter. select inter.	0.92 (1.25)	0.97 (2.06)	0.97 (2.08)	0.92 (1.25)	0.97 (2.06)	0.97 (2.08)
CPLEX	best easy: 1.53, best intermediate: 2.77, best testing: 3.03					
	Runs of ParamILS			CPU Time		
	10	25	50	1 Day	2.5 Days	5 Days
train easy select easy	1.63 (1.20)	1.63 (1.26)	1.61 (1.38)	1.63 (1.23)	1.61 (1.26)	2.36 (1.53)
train easy select inter.	**1.94** (1.54)	**2.24** (1.83)	**2.64** (1.92)	**2.00** (1.54)	**2.36** (1.83)	**2.64** (1.92)
train inter. select inter.	1.65 (1.63)	1.96 (1.88)	1.98 (1.99)	1.87 (1.71)	1.98 (1.88)	1.98 (1.99)

	Relative Win Rate (Configuration Win Rate / Default Win Rate)					
Fuego - Black	best easy: 1.04, best intermediate: 1.21, best testing: 1.22					
	Runs of ParamILS			CPU Time		
	10	15	30	50 Days	75 Days	150 Days
train easy select easy	1.08 (1.02)	1.08 (1.02)	0.94 (1.04)	1.08 (1.02)	1.08 (1.02)	0.94 (1.04)
train easy select inter.	**1.12** (1.17)	**1.13** (1.20)	**1.17** (1.21)	**1.12** (1.17)	**1.13** (1.20)	**1.17** (1.21)
train inter. select inter.	1.10 (1.10)	1.06 (1.21)	1.06 (1.21)	1.10 (1.10)	1.06 (1.21)	1.06 (1.21)
Fuego - White	best easy: 1.07, best intermediate: 1.13, best testing: 1.45					
	Runs of ParamILS			CPU Time		
	10	15	30	50 Day	75 Days	150 Days
train easy select easy	1.13 (1.05)	1.13 (1.05)	1.25 (1.07)	1.13 (1.05)	1.13 (1.05)	1.25 (1.07)
train easy select inter.	1.27 (1.08)	**1.41** (1.13)	**1.41** (1.13)	1.27 (1.08)	**1.41** (1.13)	**1.41** (1.13)
train inter. select inter.	**1.32** (1.12)	1.32 (1.12)	1.34 (1.12)	**1.32** (1.12)	1.32 (1.12)	1.34 (1.12)

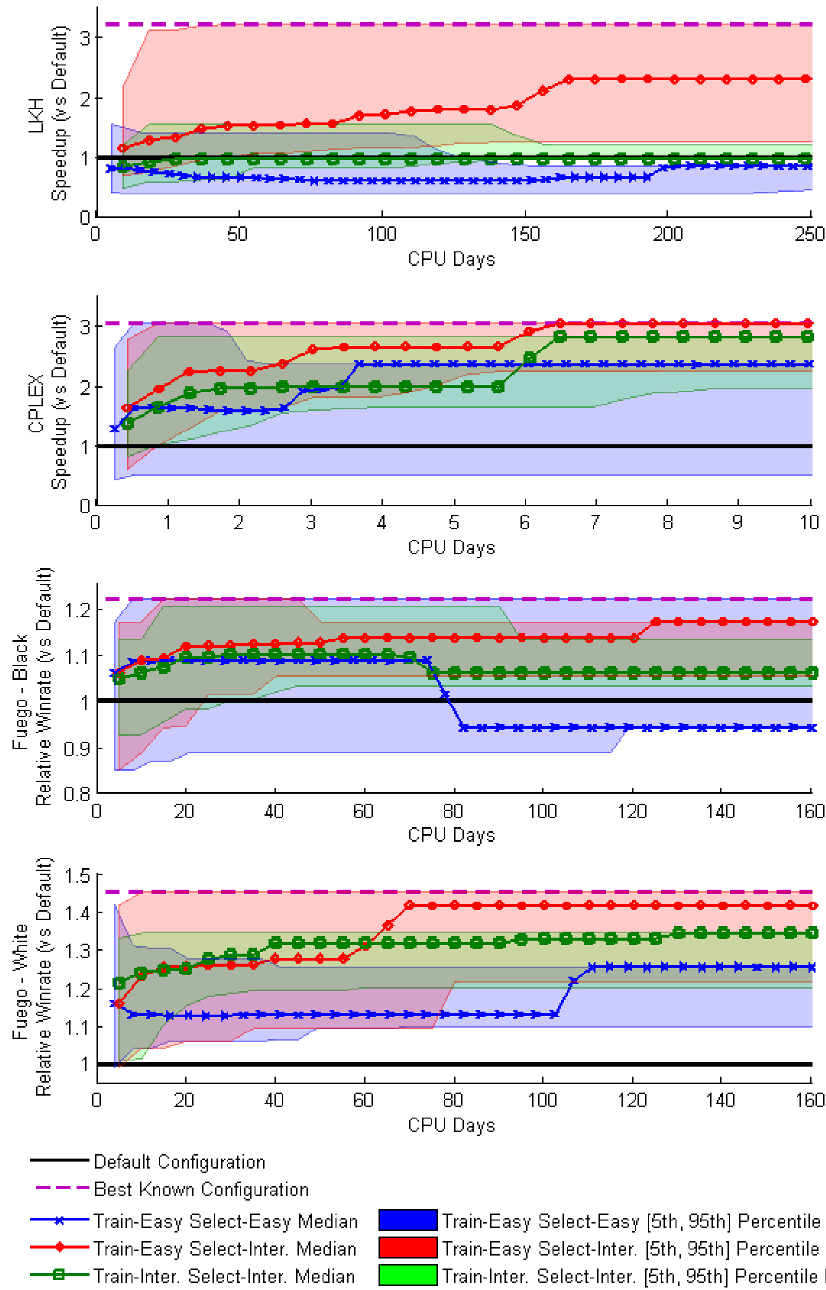

Fig. 2. Relative performance of the configurations found using the three protocols versus overall CPU time budget spent, including the median and [5th,95th] percentiles over 100 000 bootstrapped samples for every protocol as well as the quality of the default and best known configurations for reference. For all three configuration scenarios, the TE-SI protocol yields the best overall results.

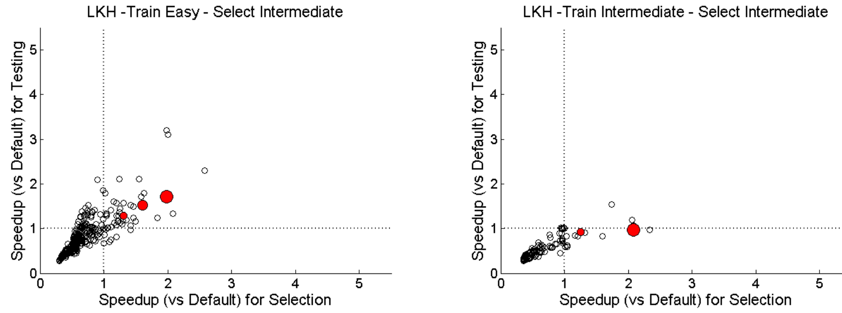

Fig. 3. Comparison between the selection and testing (PAR10) speedup, relative to the default configurations, for configurations of LKH found by 300 runs of ParamILS using TE-SI (left pane) and 100 runs of ParamILS using TI-SI (right pane). Configurations shown in the CPU time column of Table 1 are filled in and coloured red. The size of the points for these configurations corresponds to the time required to find them.

6.1 Results for configuring LKH

The TE-SI protocol was able to reliably improve upon the default configuration (see Fig. 2). The other two protocols tend to either produce configurations with quality similar to the default (TI-SI) or notably worse than the default (TE-SE).

Our bootstrap analysis reveals that for small overall time budgets, there is a 5% chance for both the TE-SE and the TI-SI protocols to produce configurations which perform better than the default (see 95th percentile curve). However, as the CPU time budget is increased to 100 CPU days and beyond, this probability decreases significantly. The reason underlying this phenomenon is apparent from Figures 1 and 3: Both protocols encounter, with some probability, configurations with excellent selection performance but poor testing performance, and as more runs of ParamILS are performed, the chances of obtaining at least one such misleading configuration increases. We note that the precise location and magnitude of the drop in 95th percentile shown here depends on the set of runs from which we obtained our bootstrapped samples and would likely be somewhat different if the entire experiment were repeated. However, we expect that drops of some magnitude are likely to occur.

6.2 Results for configuring CPLEX

This is a scenario where the standard protocol is known to be effective [12], and this result is confirmed by our results shown in Figure 2. While both protocols that select on intermediate are able to reliably found and selected good configurations, the protocol we propose generally provides the best results. For TE-SE there is still a significant ($\geq 5\%$) chance that the final configuration selected will be worse than the default; this is can be attributed to two configurations found, see Figure 4, with training speedups between 1.3 and 1.4 and testing speedups of 0.5.

Similar to the results for LKH, there is a decrease in the 95th percentile quality for configurations found using TE-SE. Again, this can be explained by the existence of misleading configurations seen in Figure 4.

6.3 Results for con guring Fuego

Like the previous scenarios, using the TE-SI protocol provides the best overall performance when configuring Fuego for either playing black or white (see Figure 2). Interestingly, our results indicate that it is much easier to improve upon the default configuration of Fuego for playing white, and the majority of configurations found by all three protocols for playing white were indeed better than the default, see Figure 2.

Similar to the other scenarios the TE-SE protocol suffered degrading performance when given additional computational resources, but surprisingly, the TI-SI protocol suffered from this as well when configuring for playing black. Looking at Figure 4, we can see that this is due to the presence of one outlier of particularly good quality (w.r.t. testing quality).

We are only presenting the scatter plots for configurations of Fuego trained to play black due to limited space. The results were qualitatively similar.

7 Conclusions and future work

In this paper we have shown that the TE-SI protocol provides benefit over the alternatives whenever it is infeasible to train directly for the intended use case of an algorithm given the available computational resources. Our simple modification to the standard protocol for using automated configurators does not require any modification to the underlying configurator and allows existing experiments to be reused. We have then demonstrated, through a large empirical study, the effectiveness of the TE-SI protocol across three different configuration scenarios. For solving MIP with CPLEX, a scenario where the standard TE-SE protocol is known to be effective, the TE-SI protocol was able to improve upon the results of the standard protocol for short configurator runs; we believe it will continue to provide benefit for longer runs using harder instance and currently investigate this hypothesis. In the other two scenarios, where the standard protocol is either unable (for solving TSP using LKH) or unlikely (for playing Go using Fuego) to yield good configurations, the TE-SI protocol is reliably able to produce better configurations than both the TE-SE and TI-SI protocols and facilitates substantial improvements over the default configurations.

We see three main avenues for future work. First, we are currently extending the analysis of our new protocol by testing it on additional configuration scenarios, including CPLEX 12.3 applied to a harder set of MIP instances, based on real-world data modeling the spread of red-cockaded woodpeckers [3], as well as Concorde [5], the state-of-the-art complete TSP solver, on TSPLIB instances. We also plan to evaluate how well our new protocol works in conjunction with other algorithm configuration procedures, in particular, the latest version of SMAC [13]. Second, we have begun to investigate the use of predictive models in improving the effectiveness of selecting configurations. Finally, we plan to apply the methods presented in this paper as well as any that result from future work to configuring new versions of Fuego for upcoming Go competitions.

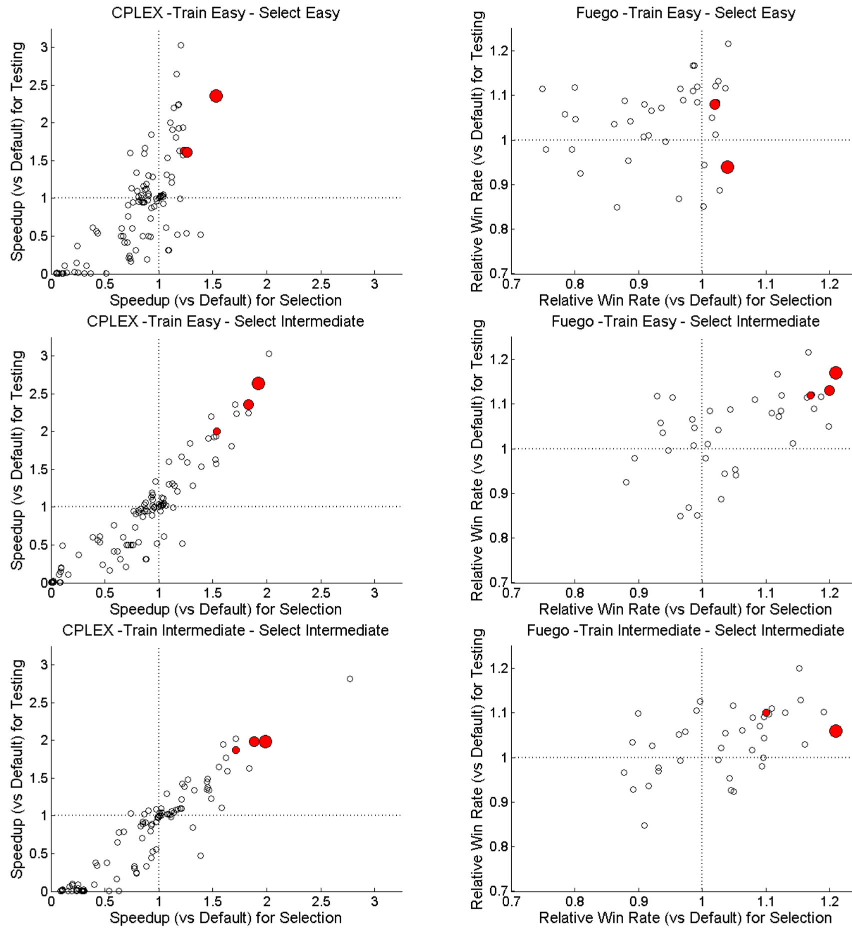

Fig. 4. Comparison between the selection and testing performance of CPLEX (left side), measuring PAR10 speedup, and Fuego (right side) trained for playing black, measuring relative winrate, found by multiple independent runs (100 for CPLEX and 40 for Fuego) of ParamILS using the TE-SE (top), TE-SI (middle) and TI-SI (bottom) protocols. Configurations shown in the CPU Time column of Table 1 are filled in and coloured red. The size of the points for these configurations corresponds to the time required to find them.

References

[1] Fuego. http://fuego.sourceforge.net/. Version visited last in October 2011.

[2] IBM ILOG CPLEX optimizer. http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/. Version visited last in October 2011.

[3] K. Ahmadizadeh, B. Dilkina, C. P. Gomes, and A. Sabharwal. An empirical study of optimization for maximizing diffusion in networks. In *Proceedings of the 16th International Conference on Principles and Practice of Constraint Programming*, CP'10, pages 514–521. Springer-Verlag, 2010.

[4] C. Ansótegui, M. Sellmann, and K. Tierney. A gender-based genetic algorithm for the automatic configuration of algorithms. In *CP-09*, pages 142–157, 2009.

[5] D. Applegate, R. E. Bixby, V. Chvátal, and W. J. Cook. Concorde TSP solver. http://www.tsp.gatech.edu/concorde.html. Version visited last in October 2011.

[6] M. Birattari, T. Stützle, L. Paquete, and K. Varrentrapp. A racing algorithm for configuring metaheuristics. In *GECCO-02*, pages 11–18, 2002.

[7] M. Birattari, Z. Yuan, P. Balaprakash, and T. Stützle. F-Race and Iterated F-Race: An Overview. In *Experimental Methods for the Analysis of Optimization Algorithms*, pages 311–336. Springer-Verlag, 2010.

[8] M. Chiarandini, C. Fawcett, and H. H. Hoos. A modular multiphase heuristic solver for post enrolment course timetabling. In *Proceedings of the 7th International Conference on the Practice and Theory of Automated Timetabling*, pages 1–6, Montréal, 2008.

[9] M. Enzenberger, M. Müller, B. Arneson R., and Segal. Fuego - an open-source framework for board games and Go engine based on Monte Carlo tree search. In *IEEE Transactions on Computational Intelligence and AI in Games*, volume 2, pages 259–270, 2010. Special issue on Monte Carlo Techniques and Computer Go.

[10] C. P. Gomes, W. Jan van Hoeve, and A. Sabharwal. Connections in networks: A hybrid approach. In *CPAIOR*, volume 5015 of *LNCS*, pages 303–307. Springer, 2008.

[11] K. Helsgaun. An effective implementation of the Lin-Kernighan traveling salesman heuristic. In *European Journal of Operational Research*, volume 126, pages 106–130, 2000.

[12] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Automated configuration of mixed integer programming solvers. In *Proc. of CPAIOR-10*, volume 6140 of *LNCS*, pages 186–202. Springer, 2010.

[13] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Proc. 5th Intl. Conference on Learning and Intelligent Optimization (LION 5)*, volume 6683 of *LNCS*, pages 507–523. Springer-Verlag, 2011.

[14] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle. ParamILS: An Automatic Algorithm Configuration Framework. In *Journal of Arti cial Intelligence Research*, volume 36, pages 267–306, October 2009.

[15] G. Reinelt. TSPLIB. http://www.iwr.uni-heidelberg.de/groups/comopt/ software/TSPLIB95. Version visited last in October 2011.

[16] D. Tompkins and H. H. Hoos. Dynamic scoring functions with variable expressions: new SLS methods for solving SAT. In *Theory and Applications of Satis ability Testing (SAT)*, 2010.

[17] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Hydra-MIP: Automated algorithm configuration and selection for mixed integer programming. In *RCRA Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion at the International Joint Conference on Arti cial Intelligence (IJCAI)*, 2011.