TRT - INRIA - ONERA

ANR-09-COSI-002 Technical Report

DESCARWIN

The Marriage of Descartes and Darwin

D2.2

Experiments using off-line methods

DOCUMENT CONTROL

	–: APR. 13, 2011	A:	B:	C:
Written by Signature	Jacques Bibaï Matthias Brendel Pierre Savéant Marc Schoenauer Vincent Vidal			
Approved by Signature	P. Savéant			

Revision index	Modifications
–	initial version
A	
B	
C	

1. PURPOSE

This document reports experiments regarding the off-line tuning of the parameters of the Divide-and-Evolve method. Two very different approaches have been used, and for each one, a published paper presenting the corresponding work is included in this document after a short presentation.

- The first Chapter presents work involving a standard *Racing* parameter tuning method, as introduced by Birratari et al. (reference [7] in the first paper). The goal is to find the best parameter setting for a class of instances, based on trials on a few instances for that class. The corresponding paper is “On the Generality of Parameter Tuning in Evolutionary Planning”, by Jacques Bibaï, Pierre Savéant, Marc Schoenauer, and Vincent Vidal, and was presented at the ACM-GECCO conference in July 2010 in Seattle (pp 241-248 of the proceedings).
- The second Chapter introduces an original method for instance-based parameter tuning, where a model of difficulty w.r.t. given algorithm is learned on some sample instances, and used thereafter on unknown instances to predict which are the best parameters for this instance. The corresponding paper has been accepted for poster presentation at the ACM-GECCO 2011 in Dublin next July (proceedings not yet available).

CONTENTS

2. ON THE GENERALITY OF PARAMETER TUNING IN EVOLUTIONARY PLANNING

The paper presented in this Chapter presents off-line parameter tuning applied to DaE using the well-known Racing method [7][1]. The goal of the Racing method is to find the best setting of parameters among a finite pre-defined set of possible parameter settings. The brute force way would be to run the several instances of the algorithm with all available settings, and to use standard statistical analysis (e.g. ANOVA) to find out the best setting. The idea of Racing is to start running only a small number of runs per available setting, and to eliminate as early as possible the settings that obviously perform poorly (according to some statistical test). The process stop when either only one parameter setting remains, or when a pre-defined number of runs have been run for all settings - and standard ANOVA can then be used on the remaining settings. Typical savings compared to the brute force are from 50% to 90% of computational effort.

The other issue that arises when dealing with programs like DaE that optimize instances of a given problem is that of the generality of parameter tuning. The best results are of course obtained when the parameters are tuned anew for each instance that is to be optimized. However, this process is very costly indeed: even when using Racing, one parameter tuning process means represents several hundreds of complete runs. Hence the goal of parameter tuning in this situation is to find some parameter settings that give quasi-optimal results for several instances, or classes of instances. The closer the instances, the higher the chances that this is possible. Hence good candidates for such classes where all instances share some characteristics are the problem domains as defined in the IPC competitions: each domain contains around 30 instances that share the same objects types and actions. Picking up a few instances per domain, and running the Racing based on those instances only hopefully results in parameter settings that generalize to the whole domain. Finally, the ideal parameter setting should perform quasi-optimally on all instances of all domains. Similarly to what can be done within a domain, Racing is then done based on the results of the parameter sets on several instances of several domains.

The following paper contains the analysis of the results of the different options (tuning per instance, tuning per domain, global tuning). As expected, the more global the tuning, the worse the results. However, the results of the global tuning, frozen as the default parameter setting for DaE, allowed us to generate the comparative results of DaE with the state-of-the-art planners published in [4] that received the Silver Medal at the *Human Competitive Results* at ACM-GECCO conference in 2010.

[1] The citations refer to the references of the following paper.

On the Generality of Parameter Tuning in Evolutionary Planning

Jacques BIBAI [1,2]
Thales Research & Technology[2]
Palaiseau, France
firstname.lastname@thalesgroup.com

Pierre SAVÉANT
Thales Research & Technology[2]
Palaiseau, France

Marc SCHOENAUER
Projet TAO, INRIA Saclay, & LRI-CNRS, Univ. Paris Sud[1] & Joint lab Microsoft-INRIA
Orsay, France
firstname.lastname@inria.fr

Vincent VIDAL
ONERA – DCSD
Toulouse, France
Vincent.Vidal@onera.fr

ABSTRACT

Divide-and-Evolve (DaE) is an original "memeticization" of Evolutionary Computation and Artificial Intelligence Planning. However, like any Evolutionary Algorithm, DaE has several parameters that need to be tuned, and the already excellent experimental results demonstrated by DaE on benchmarks from the International Planning Competition, at the level of those of standard AI planners, have been obtained with parameters that had been tuned once and for-all using the Racing method. This paper demonstrates that more specific parameter tuning (e.g. at the domain level or even at the instance level) can further improve DaE results, and discusses the trade-off between the gain in quality of the resulting plans and the overhead in terms of computational cost.

Categories and Subject Descriptors

I.2.8 [**Computing Methodologies**]: Artificial Intelligence Problem Solving, Control Methods, and Search

General Terms

Algorithms

Keywords

AI Planning, Memetic Algorithms, Parameter Tuning, Racing

1. INTRODUCTION

Parameter tuning is known as one of the main Achilles' heel of Evolutionary Algorithms (together with their computational cost). Whereas the efficiency of EAs has been demonstrated on several application domains (see for instance [33]), their successes were in general obtained through a tedious and time consuming parameter tuning phase. Furthermore, the newcomer to the field cannot benefit from theoretical guidance or recognized guidelines, and is then tempted to use off-the-shelf parameters, i.e. either default parameters of the framework he is using, or parameter values given in the literature for problems that resemble his.

Parameter setting, however, has today become an important field of research [19]. Following [9], one distinguishes between parameter tuning, that is done off-line, before running the algorithm, and parameter control, that is performed during the run, either based on the behavior of the algorithm (referred to as parameter adaptation), or relying on random modifications of the parameters together with the idea that fitness-based selection will also select adapted parameters (referred to as parameter self-adaptation). This paper is concerned with parameter tuning, i.e. off-line setting of the parameters of an Evolutionary Algorithm, in the domain of AI planning.

Several methods have been proposed for parameter tuning since Grefenstette's pioneering work [14]. However, they all face the same generalization issue: can a parameter set that has been optimized for a given problem be successfully used to some other problem? The answer of course depends on the similarity between both problems – and the issue then is to estimate the similarities between different optimization problems. However, even restricted to some precise optimization domain (like AI Planning), there are very few examples today of meaningful similarity measures between optimization problems, or, alternatively, of sufficiently precise and accurate features that would allow the user to describe the problem at hand with sufficient accuracy so that the optimal parameter set could be learned from this description, and carried on to other problems with similar description. The one exception we are aware of is the work of Hoos and co-authors in the SAT domain [17]: based on half a century of SAT work, and hundreds of papers, many relevant features have been gathered. Extensive parameter tuning on several thousands of instances have allowed the authors to learn a meaningful mapping between several parameteriza-

GECCO'10, July 7–11, 2010, Portland, Oregon, USA.

tion of a given algorithm and the resulting performance of the algorithm (in terms of time to solution for satisfiable SAT problems). Such mapping allows the user to come up with a good, if not optimal, set of parameters by applying this mapping to the features that can be easily computed for each unknown instance she/he has to solve.

This paper is concerned with such issues in the domain of the Artificial Intelligence (AI) Planning problems. An AI planning task written in PDDL (more details in Section 2.1) is specified by a *domain*, that defines the predicates and the actions, and an *instance*, that instantiates the objects, and specifies the initial state and a list of goals to reach. A solution is a plan, or a sequence of actions, such that when applied to the initial state it leads the system into a state where all goal atoms are true. Hence, for a given domain, there can exist several instances sharing the same predicates and actions, but differing either by the number of objects, or the initial and goal states.

Unfortunately, there does not exist any set of features for the AI Planning problem that are sufficient to describe the characteristics of a planning task, like mentioned above for the SAT domain [17]. The goal of this paper is hence less ambitious, though it can be seen as a first step in the direction of automatic parameter setting for the Evolutionary AI planner considered here. We will investigate the generality of behavioral parameters, i.e. parameters of the algorithm that directly impact on its behavior during the run (mainly the parameters related to the different variation operators, from their application rates to their internal parameters). We will also consider some structural parameters, namely here the set of predicates that are used to build the candidate solutions in the Evolutionary planner at hand, and analyze whether this set can be reduced, based on the analysis of previous runs, in order to reduce the size of the search space, and, hopefully, the computational cost of the optimization, without degrading the quality of the resulting solutions.

More practically, the experimental investigation proposed here will use 3 different domains of the AI Planning problem that have been proposed in the past International Planning Competitions (of 3 different types, see Section 4.1). Each domain contains a series of instances. Racing [7] will be conducted at a global level (aggregating quality from one instance from each domain), at the domain level (aggregating quality of the biggest instance of the domain), and at the instance level (an optimal parameter set will be determined for each instance). The way the performance of the algorithm decreases when going from instance to domain to global levels will give indications about both the homogeneity of the different domains and instances inside a domain, and the robustness of the parameter setting for the Evolutionary Planner at hand. Furthermore, the very good results that have already been obtained by DaE on those instances compared with the state-of-the-art have used a single parameter set (i.e. the result of global racing) [4]. The results presented here will assess that there is still room for improvement for the Evolutionary Planner DaE.

The paper is organized as follows: domain independent AI Planning is briefly introduced in Section 2.1. Section 2 presents DaE, the specific Evolutionary Planner. Section 3 rapidly recalls the basics of the Racing procedure that has been used here, as well as the experimental protocol. Experimental results are presented and analyzed in Section 4, comparing the performances of the parameter sets obtained with different levels of racing, and comparing the variation among the sets. As usual, Section 5 concludes the paper by sketching directions for further research.

2. DIVIDE AND EVOLVE

2.1 AI Planning

An Artificial Intelligence (AI) planning task is specified by the description of an initial state, a goal state, and a set of possible actions. An action can be applied only if certain conditions in the current state are met, and modifies the current state when applied. A solution to a planning task is an ordered set of actions, whose execution from the initial state transforms it into a state that includes the goal state. The quality criterion of a plan depends on the type of available actions: number of actions in the simplest case (aka STRIPS domain); total cost for actions with cost; total makespan for durative actions which, in addition, may temporally overlap.

Domain-independent planners rely on the Planning Domain Definition Language (PDDL) [22], inherited from the STRIPS model [10], to standardize and represent a planning task. The language has been extended for representing temporality and action concurrency in PDDL2.1 [11]. The history of PDDL is closely related to the different editions of the International Planning Competitions (IPCs `http://ipc.icaps-conference.org/`), and the problems submitted to the participants are still the main benchmarks in AI Planning (see Section 4.1).

The description of a planning task splits into two separate parts: the generic *domain* on the one hand and a specific *instance* scenario on the other hand. The domain definition specifies object types, predicates and actions which capture the possible state changes, whereas the instance scenario declares the objects of interest, gives the initial state and provides a description of the goal. A *state* is described by a set of atomic formulae, or atoms. An atom is defined by a predicate symbol from the domain definition, followed by a list of object identifiers: $(PREDICATE\_NAME\ OBJ_1\ ...\ OBJ_N)$.

The initial state is complete, i.e. it gives a unique status of the world, whereas the goal might be a partial state, i.e., it can be true in many different (complete) states. An action is composed of a set of preconditions and a set of effects, and applies to a list of variables given as arguments, and possibly a duration or a cost. Preconditions are logical constraints which apply domain predicates to the arguments and trigger the effects when they are satisfied. Effects enable state transitions by adding or removing atoms.

A solution to a planning task is a consistent schedule of grounded actions whose execution in the initial state leads to a state that contains one goal state, i.e., where all atoms of the problem goal are true. A planning task defined on domain D with initial state I and goal G will be denoted $\mathcal{P}_D(I, G)$ in the following.

2.2 Evolutionary AI Planning

Early approaches to AI Planning using Evolutionary Algorithms directly handled possible solutions, i.e. possible plans: an individual is an ordered sequence of actions [28, 23, 31, 32, 8]. However, as it is often the case in Evolutionary Combinatorial optimization, those direct encoding approaches have limited performance in comparison to the

traditional AI planning approaches, and hybridization with classical methods had been the way to success in many combinatorial domains, as witnessed by the fruitful emerging domain of memetic algorithms [15]. Along those lines, though relying on an original "memeticization" principle, a novel hybridization of Evolutionary Algorithms (EAs) with AI Planning, termed *Divide-and-Evolve* (DAE) has been proposed [26, 27]. For a complete formal description, see [4].

In order to solve a planning task $\mathcal{P}_D(I, G)$, the basic idea of DAE is to find a sequence of states $S_1, \ldots, S_n$, and to use some embedded planner to solve the series of planning tasks $\mathcal{P}_D(S_k, S_{k+1})$, for $k \in [0, n]$ (with the convention that $S_0 = I$ and $S_{n+1} = G$). The generation and optimization of the sequence of states (S_i) is driven by an evolutionary algorithm, and we will now describe its main components: the problem-specific representation of individuals, fitness, and variation operators.

2.3 DAE Representation

A state is a list of atoms built over the set of predicates and the set of object instances. However, searching the space of complete states would result in a rapid explosion of the size of the search space. Moreover, goals of planning problem need only to be defined as partial states. It thus seems more practical to search only sequences of partial states, and to limit the choice of possible atoms used within such partial states. However, this raises the issue of the **choice of the atoms** to be used to represent individuals, among all possible atoms. The result of the previous experiments on different domains of temporal planning tasks from the IPC benchmark series [6] demonstrates the need for a very careful choice of the atoms that are used to build the partial states. The method used to build the partial states is based on an estimation of the earliest time from which an atom can become true. Such estimation can be obtained by any admissible heuristic function (e.g h^1, h^2... [16]). The possible start times are then used in order to restrict the candidate atoms for each partial state. A partial state is built at a given time by randomly choosing among several atoms that are possibly true at this time. The sequence of states is then built by preserving the estimated chronology between atoms (**time consistency**). Heuristic function h^1 has been used for all experiments presented here. However, these restrictions may still contain a large number of atoms, and it might be possible to further restrict this list only allowing atoms that are built with a restricted set of predicates. Manual choice had been used in the early versions of DAE [3]. However, it can be expected that such structural parameters can be learned by post-mortem analyzes of different runs of DAE on several problems of the same domain. This will be investigated in Section 4.

Nevertheless, even when restricted to specific choices of atoms, the random sampling can lead to inconsistent partial states, because some sets of atoms can be *mutually exclusive*[1] (`mutex` in short). Whereas it could be possible to allow `mutex` atoms in the partial states generated by DAE, and to let evolution discard them, it seemed more efficient to try to a priori forbid them. In practice, it is not possible to decide if several atoms are `mutex` unless solving the complete problem. Nevertheless, binary `mutexes` can be approximated with a variation of the h^2 heuristic function [16] in order to build quasi pairwise-mutex-free states (i.e., states where no pair of atoms are mutex).

An individual in DAE is hence represented as a variable-length ordered time-consistent list of partial states, and each state is a variable-length list of atoms that are not pairwise `mutex`. Furthermore, all operators that manipulate the representation (see below) maintain the chronology between atoms and the local consistency of a state, i.e. avoid pairwise mutexes.

2.4 Initialization and Variation Operators

The **initialization** of an individual is the following: first, the number of states is uniformly drawn between 1 and the number of estimated start times (see Section 2.3); For every chosen time, the number of atoms per state is uniformly chosen between 1 and the number of atoms of the corresponding restriction. Atoms are then chosen one by one, uniformly in the allowed set of atoms, and added to the individual if not `mutex` with other atoms that are already there.

A 1-point **crossover** is used, adapted to variable-length representation in that both crossover points are independently chosen, uniformly in both parents. It is applied with probability p_{cross} to the individuals after selection.

After a possible crossover, an individual has a probability p_{mut} of being mutated. Four different mutation operators have been designed, and once an individual has been chosen for mutation (according to a population-level mutation rate), the choice of which mutation to apply is made according to user-defined relative weights (see Section 4.1). Because an individual is a variable length list of states, and a state is a variable length list of atoms, the mutation operator can act here at two levels: at the individual level by adding (**addState**) or removing (**delState**) a state; or at the state level by adding (**addAtom**) or removing (**delAtom**) some atoms in the given state. The choice between those operators is governed by user-defined relative *weights* ($w_{addStation}$, $w_{delStation}$, $w_{addAtom}$, $w_{delAtom}$). Furthermore, mutation addAtom also modifies every other atom of the state it is applied to with probability p_c, and, when adding a state, the possible new atoms are those that are possibly true in a time interval of radius r around the estimated time of the partial state at hand. r takes integer values, being an index in the array of possible times given by heuristic h^1 (see Section 2.3.

2.5 Fitness and Embedded Planners

In DAE, the fitness of a list of partial states $S_1, \ldots, S_n$ is computed by repeatedly calling an external 'embedded' planner to solve the sequence of problems $\mathcal{P}_D(S_k, S_{k+1})$, $\{k = 0, \ldots, n\}$. Any existing planner can be used, and the early versions of DAE used the optimal planner CPT [30]. However, recent results [5] have demonstrated that much better results can be obtained in a more robust way by using a suboptimal planner, namely YAHSP [29], a lookahead strategy planning system for STRIPS planning which uses the actions in the relaxed plan to compute reachable states in order to speed up the search process.

For any given k, if the chosen embedded planner succeeds in solving $\mathcal{P}_D(S_k, S_{k+1})$, the final complete state is computed by executing the solution plan from S_k, and becomes the initial state of the next problem. If all problems, $\mathcal{P}_D(S_k, S_{k+1})$ are solved by the chosen embedded planner,

[1] Several atoms are mutually exclusive when there exists no plan that, when applied to the initial state, yields a state containing them all.

the individual is called *feasible*, and the concatenation of all solution plans for all $\mathcal{P}_D(S_k, S_{k+1})$ is a global solution plan for $\mathcal{P}_D(S_0 = I, S_{n+1} = G)$. However, in the case of temporal planning, this plan can in general be optimized by rescheduling some of its actions, in a step called *compression* in order to get a better makespan (see [27] for detailed discussion). The quality of the compressed plan defines the fitness of a feasible individual.

On the other hand, as soon as the chosen embedded planner fails to solve one $\mathcal{P}_D(S_k,\ S_{k+1})$ problem, the following problem $\mathcal{P}_D(S_{k+1}, S_{k+2})$ cannot be even tackled by the chosen embedded planner, as its initial state is in fact partially unknown. All such plans receive a penalty inversely proportional to the number of solved subproblems, and such that the fitness of any infeasible individual is higher than that of any feasible individual.

Finally, in order to avoid the embedded planner to be stuck with subproblems that are in fact more difficult than the original one, YAHSP was constrained not to use more than a given number of nodes when solving any of the subproblems: first, a very large bound (e.g. 100000) is set when evaluating the initial population. The bound for the remaining of the run is then chosen as the median of the actual number of nodes that have been used to find the solutions during these evaluations.

3. PARAMETER TUNING FOR DAE

Two types of parameters are distinguished here: *structural* parameters act at the representation level, and include here the choice of the predicates that are used to build the intermediate states (see Section 2.3); Behavioral parameters directly impact on the way the optimization proceeds, and include the population size, the selection procedure and its parameters, plus all parameters related to the variation operators. However, the Darwinian-related parameters of DAE had been fixed after some early experiments [26, 27] (see Section 4.1), and only the 8 parameters related to the variation operators that have been described in Section 2.4 have been tuned using Racing.

Whereas tuning behavioral parameters pertains to standard Parameter Tuning procedure (Racing has been used here), problem-specific methods are required for the structural parameters. This Section details both tuning procedures in turn.

3.1 Learning Predicates Across Runs

The set of predicates that are used to describe the partial states of intermediate goals (see Section 2.3) is an important component of the representation, and has a large impact on the size of the search space. Hence pruning it as much as possible can become a crucial issue in very large domains. In order to assess the usefulness of a possible *Learning across runs* technique, where, within a given domain, some runs could help identifying the useful predicates, the following statistics were gathered during the experiments. 50 runs were run on the first 11 instances of each domain, using the parameters learned at the domain level. All atoms of the 11 $\times$ 50 best plans were gathered, and two figures were computed for each predicate: First, the frequency of appearance, i.e. the proportion of those optimal plans that did contain this predicate (number in $[0, 1]$ for each predicate, 1 would mean that this predicate was present in every solution plan);

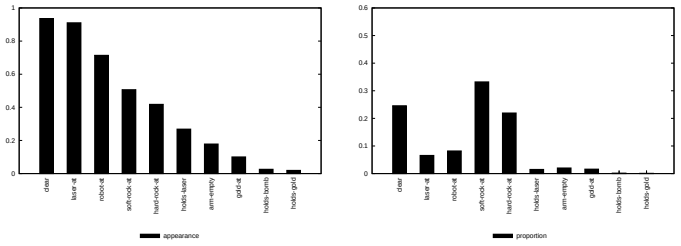

Figure 1: Predicate statistics: Frequency of occurrence (left) and proportion in all atoms (right) for `gold-miner` domain. The predicates are in the same order on the x-axis of both plots.

Second, the proportion of atoms that did contain this predicate – all those proportions sum up to 1.

Based on those statistics, some clustering method was applied to this dataset. Relational analysis [2] with a threshold of similarity equal to 0.5 was chosen: initiated and developed at IBM in the 70s [20], relational analysis does not require to arbitrarily choose the number of clusters to be discovered. After this analysis, the predicates that belong to classes with more than 2 elements are retained, with the expectation that restraining the choice of predicates for the resolution of the other instances (not the first 11 ones, that were used to identify those predicates) would speed up the search (because the search space is smaller) while maintaining the same quality of the solution.

This approach was tried on several domains, and Figure 1 displays both statistics described above for the `gold-miner` domain, using the experimental settings described in next Section. Both statistics obviously capture different criteria of importance for the predicates. Unfortunately, the results obtained by DAE when using the restricted set of predicates chosen according to either of those statistics were not better, and sometimes even worse, than the results of DAE using the complete set of predicates. Furthermore, even when the results were similar in quality, they were not obtained significantly faster. Based on those experiments, it is not clear that a restricted set of predicates can be learned, and at least it cannot be learned as proposed here: the automatic choice of a restricted set of predicates is still an open issue, subject of further work.

3.2 Racing Behavioral Parameters

Since the early days of Evolutionary Algorithms [14], researchers have tried to optimize the control parameters of their favorite algorithms. Classical statistical methods like ANOVA (Analysis of Variance), based on some extensive DOE (Design Of Experiment) can be used out-of-the-box: a finite number of parameter sets is chosen (e.g. a factorial design that includes all possible combinations of a finite number of values for each parameter), and a given number of runs is run for each set. Standard ANOVA test indicates whether some statistically significant difference exists among all the sets, and pairwise tests then designate the best set.

Originally proposed for solving the model selection issue in Machine Learning [21], Racing techniques were introduced in Evolutionary context [7] in order to decrease the computational cost of such naive approach by rapidly focusing the search on the most performing parameter configurations. The basic idea of Racing techniques is to identify, with a given statistical confidence level, the poorly-performing

parameter configurations after only a few runs, and to go on running only the promising configurations: after each run, all configurations are tested against the best one, and the ones that are significantly worse are simply discarded. Such cycle execution-comparison-elimination is repeated until only one parameter configuration remains, or the total number of experiments has been run.

The efficiency of such technique highly depends on the statistical test used for the comparison. Because no assumption can be made about the distribution of the results (e.g. normality), the most popular races [7, 34, 35] are based on the non-parametric Friedman's two-way analysis of variances by ranks. Furthermore, given the statistical test, the user must set two parameters for Racing: the number of initial runs before starting the comparisons, and the confidence level of the test.

The performance used for the comparison is generally the best fitness reached for a given stopping criterion. Here, in order to slightly favor the algorithms that run faster, a secondary component was added to the performance measure: the fitness takes only integer values (however much the type of problem, see Section 4.1), and some penalty in [0,1] was hence added to the fitness. This penalty is proportional to the computational time of the algorithm (normalized by the maximum allowed time).

4. EXPERIMENTAL RESULTS

4.1 Experimental Setting

Previous experiments demonstrate that DAE can greatly increase the performance of an encapsulated suboptimal planner, both in terms of coverage and solution quality, making it competitive with state-of-the art planners [4] even when it have used a single parameter set for all problems.

In order to show that there is still room for improvement for the Evolutionary Planner DAE, experiments have been run on 3 domains, and represent 3 different types of problems: `gold-miner` is a STRIPS problem (the quality of a plan is the number of actions) from the IPC6 learning track, `openstacks` uses actions with costs (the quality of a plan is the total cost) of the IPC6 sequential satisficing track, and `zeno` is a temporal problem (the quality of a plan is the total makespan) from IPC3. All are available from IPC web site at `http://ipc.icaps-conference.org/`.

Furthermore, in addition to comparing the results of the different parameter sets obtained by the different parameter tuning procedures described in Section 3, DAE results have been compared to those of the best learner in their category: the optimal CPT [30] for STRIPS domain, LPG [12, 13] for temporal domain, and LAMA [25] (updated version kindly given by the authors) for actions with costs. As is the case for CPT, however, those learners were given a strict 30min time limit. In particular, this explains why the best learner is not always CPT, which is an optimal learner, and could find the optimal value for most instances if given enough time (though it cannot solve the largest `zeno` instances whatever the time it is given).

DAE has been implemented within the Evolving Objects framework (`http://eodev.sourceforge.net/`), an Open Source, STL-based, C++ Evolutionary Computation library. The fixed *evolution engine* is a (100+700)-ES: 100 individuals generate 700 offsprings without selection. The survival selection is performed among those 800 individuals using a *deterministic tournament* of size 5. For all runs, the **stopping criterion** is the following: After at least 10 generations, evolution is stopped if no improvement of the best fitness in the population is made during 50 generations, with a maximum of 1000 generations. Furthermore, all runs were allowed a maximum CPU time of 30mn (running on 3.4 GHz cores).

For the Racing procedure (see Section 3.2), the initial number of runs was set to 11 (the lowest number for significance of the statistical test), and the confidence level to 0.025 (strong constraint for the rejection of equality hypothesis between two parameter configurations). The racing process was stopped after at most 50 runs.

For the 8 behavioral parameters (Section 2.4), 2 values were tried, giving a total of 256 different parameter configurations (more configurations would have resulted in too long experiments). The following values were used: $p_{cross} = 0.2$ or 0.6, $p_{mut} = 0.6$ or 0.8, $w_* = 1$ or 3, the relative weights of the 4 mutation operators, $r = 1$ or 2, the tolerance radius for time consistency, and $p_c = 0.2$ or 0.8, probability to modify an atom in a state undergoing mutation.

The global racing was performed by aggregating the performances over one instance per domain, namely `goldminer30`, `openstacks21`, and `zeno14`. For the racing per domain, the instance with highest index (supposedly the most difficult one) was chosen, namely `goldminer30`, `openstacks30`, and `zeno20`.

4.2 Best Behavioral Parameters

Whereas $336 \times 50 = 12080$ runs would have been run by a full factorial DOE, the global racing needed only 4351 runs, and racing per domain 4013, 4021 and 3966 runs for respectively `gold-miner`, `zeno simple time`, and `openstacks` domains. the full per instance racing required on average 5259, 5817 and 5473 runs for respectively `gold-miner`, `zeno simple time`, and `openstacks` domains. However, those averages hide large variations: paradoxically, all easy instances required the maximum number of runs (12080), as several parameter configurations could solve the instance to optimum. This was the case for instances 1-20 for `gold-miner`, 1-12 for `zeno` and 1-18 for `openstacks`. Racing did save around 65% CPU time for the global or per-domain versions, and little less for the racing per instance - though we could have saved doing racing on the easy instances, as the comparative results below will confirm.

Regarding the best parameter configurations, Figure 2 displays them all, on 8 lines: the horizontal line represents the value obtained by the global racing, and only the values that differ from this one are marked for each instance (column) or the racing per domain (last column of each plot). There seems to be a large consensus on the highest value for p_c (0.8, second line from bottom), that matches the results of the global racing and all 3 domain racings. There is, too, a reasonable consensus for choosing $r = 2$ (bottom line) for `gold-miner` and `zeno` domains, matching the result of the global racing, too, but `openstacks` racings prefered the other value. On the other extreme, the values retained by the instance-racings for p_{cross}, p_{mut}, $w_{delState}$, $w_{delAtom}$, and $w_{addState}$ (lines 3-7 from bottom) seem to contradict those of the global racing, while matching those of the domain-racing only very partially. It seems that even when restricting the number of possible values of the behav-

ioral parameters to 2, different instances even belonging to the same domain are globally quite heterogenous for DaE.

4.3 Comparing Racing Procedures

Figure 3 displays the box-plots of the 50 runs for each domain (top to bottom), each racing level (left to right), and each instance (the x-axis). Additionally, the results of YAHSP alone (the embedded planner), as well as the results of the best opponent in its category. The numerical values of the means and minimal values reached are also given, for the sake of completeness, in the large unlabeled table (for space reasons) at the end of the paper.

An immediate conclusion is the confirmation that DaE outperforms by far YAHSP alone (as already demonstrated in [5]). A second obvious conclusion is that the performances of DaE when doing one racing per instance are better than the other racing options. However, this is particularly true for the most difficult instances (the instances with the highest index, at least), and for domains `zeno` and `openstack`, whereas `gold-miner` seems easier to solve to optimality, or near-optimality, with either global or per-domain racing: only slight differences for few instances can be reported for the best achieved values. However, the variation among runs, as witnessed by the height of the box-plots of Figure 3, are clearly smaller with racing per instance... some difference that is not so large on both other domains. Surprisingly, on several instances (e.g. `zeno 14, 15, 18` and `openstacks 24, 25, 28`, the global racing found better results (lower minimum, similar median) than the per-domain racing. This seems to indicate that those domains are rather heterogeneous in difficulty for the large instances.

Finally, one good news, on the other hand, is that DaE is able to find equal or better results than its best opponent in the experimental conditions chosen here, as witnessed by the last column of the final table: for some large instances, both LAMA and LPG, at least when limited to the same harsh 30min constraint that was imposed here, are frequently outperformed by DaE – at the cost of an instance-based racing to tune DaE parameters.

5. CONCLUSION

DaE is an original "memeticization" of Evolutionary and planning algorithms in the area of AI Planning. A lot of progress has been made since its original inception [26, 27], and it has now reached a level of performance that is able to compete with the best state-of-the-art planners even when it uses one single parameter configuration for all problems [4]. In this paper, however, the results on three different types of IPC problems showed that the quality of the results obtained with DaE can be further improved with more specific parameter tuning, although these results were not uniform over all tested domains. The practicality of such approach remains however questionable, as racing per instance requires to solve around 5000 times the instance with different parameter configurations. Racing per domain is an alternative whenever the user has to repeatedly solve new instances from the same domain. However, referring to the values retained by the different procedues (as discussed in Section 4.2), the different instances even of the same domain require different parameter settings, and the costly instance-based racing seems to be mandatory to get the best restults on the large and difficult instances.

Nevertheless, there is still room for improvement in tuning DaE's parameters. The choice of the set of parameter configurations for the racing is still an open issue. Furthermore, it was limited here by the CPU cost, and only 256 different parameter configurations could be tried. There are other alternatives to racing, that have the advantage that they actually optimize the parameters, i.e., are not restricted to a pre-defined set of configurations.

For instance, the Sequential Parameter Optimization method (SPO) [1] alternatively tries to build a model of the performance of the algorithm as a function of the parameters using Gaussian Kernels. Some improvements to SPO have already been proposed [18], that reduce its computational overhead and increase its accuracy. However, it remains to be compared to Racing for the Evolutionary Planning with DaE. Also, recently, the REVAC method has been proposed [24], that uses as a meta-EDA (Estimation of Distribution Algorithm) to optimize the parameters of an EA, but also estimates the relevance of each parameter at hand. This could confirm some results obtained here, where some parameters seem irrelevant because most racing results proposed the same value.

But the ultimate step in the direction proposed in this paper would be to design a completely automated procedure for parameter tuning, that would not require extensive runs that are unaffordable in real world situations for instance. The way toward this grail probably lies in Adaptive or Self-Adaptive techniques, yet to be proposed outside the continuous domain where such techniques have now reached maturity [9].

6. REFERENCES

[1] T. Bartz-Beielstein, C. Lasarczyk, and M. Preuss. Sequential parameter optimization. In B. McKay, editor, *Proc. CEC'05*, pages 773– 780. IEEE Press, 2005.

[2] H. Benhadda and F. Marcotorchino. L'analyse relationnelle pour la fouille de grandes bases de données. *In Revue des Nouvelles Technologies de l'Information*, RNTI-A-2, 2007.

[3] J. Bibai, P. Savéant, M. Schoenauer, and V. Vidal. DAE : Planning as Artificial Evolution (Deterministic part). At International Planning Competition (IPC) `http://ipc.icaps-conference.org/`, 2008.

[4] J. Bibai, P. Savéant, M. Schoenauer, and V. Vidal. An evolutionary metaheuristic based on state decomposition for domain-independent satisficing planning. In *ICAPS 2010*, pages 18–25. AAAI press, 2010.

[5] J. Bibai, P. Savéant, M. Schoenauer, and V. Vidal. On the benefit of sub-optimality within the divide-and-evolve scheme. In P. Cowling and P. Merz, editors, *EvoCOP 2010*, number 6022 in Lecture Notes in Computer Science, pages 23–34. Springer-Verlag, 2010.

[6] J. Bibai, M. Schoenauer, and P. Savéant. Divide-And-Evolve Facing State-of-the-Art Temporal Planners during the 6^{th} International Planning Competition. In C. Cotta and P. Cowling, editors, *Ninth European Conference on Evolutionary Computation in Combinatorial Optimization (EvoCOP 2009)*, number 5482 in Lecture Notes in Computer Science, pages 133–144. Springer-Verlag, 2009.

[7] M. Birattari, T. Stützle, L. Paquete, and K. Varrentrapp. A Racing Algorithm for Configuring Metaheuristics. In *GECCO '02*, pages 11–18. Morgan Kaufmann, 2002.

[8] A. H. Brié and P. Morignot. Genetic Planning Using Variable Length Chromosomes. In *Proc. ICAPS*, 2005.

[9] A. E. Eiben, Z. Michalewicz, M. Schoenauer, and J. E. Smith. Parameter control in evolutionary algorithms. In Lipcoll et al. [19], chapter 2, pages 19–46.

[10] R. Fikes and N. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 1:27–120, 1971.

[11] M. Fox and D. Long. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *JAIR*, 20:61–124, 2003.

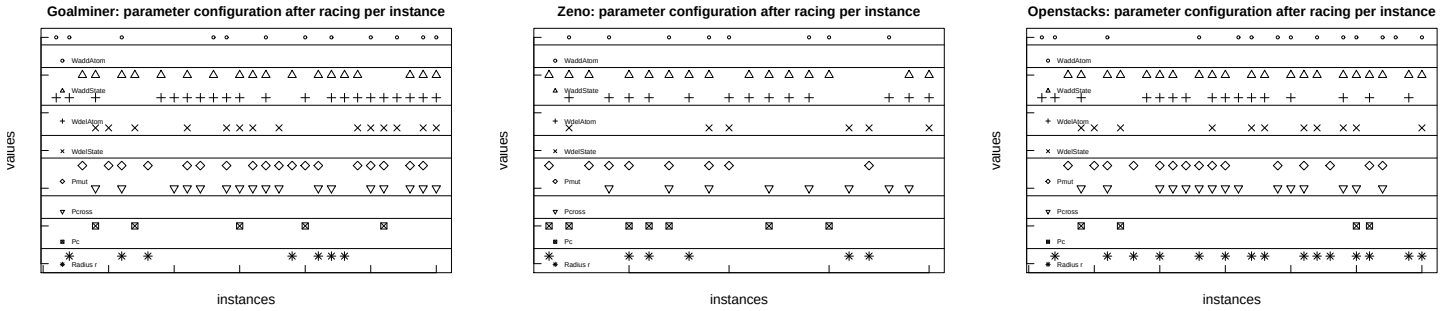

Figure 2: Best parameter configurations after racing per instance, for `gold-miner`, `zeno` and `openstacks` domains. From top to bottom, $w_{addAtom}$, $w_{addState}$, $w_{delAtom}$, $w_{delState}$, p_{mut}, p_{cross}, p_c, r, and each parameter could take 2 values (see text). For each of those parameters, the horizontal line represents the value retained by the global racing, the last column shows that retained by the racing per domain, and each other column is the value obtained from racing on a single instance.

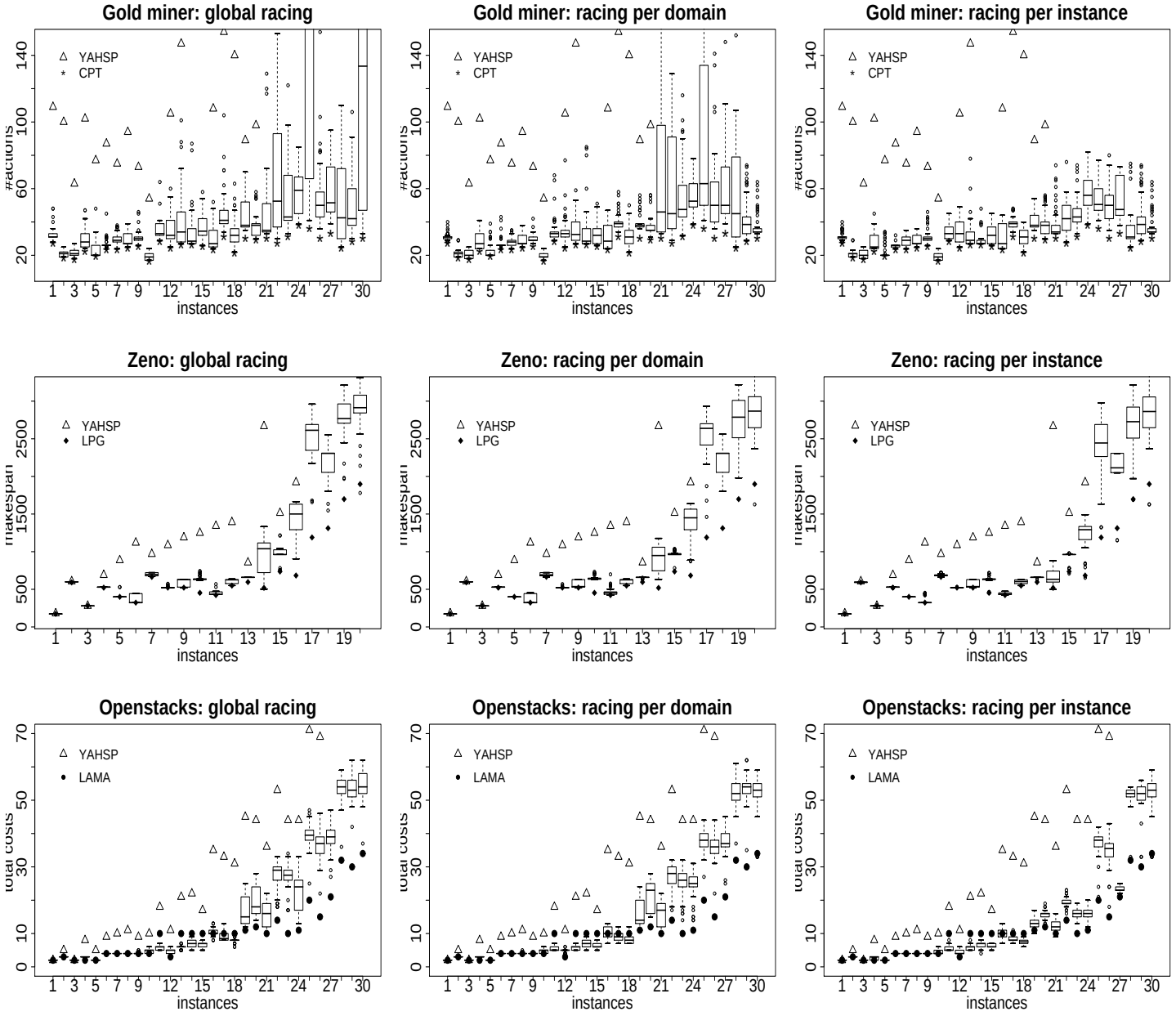

Figure 3: Results for DAE on `gold-miner`, `zeno` and `openstacks`, together with the value found by YAHSP alone (△), and the value from the best competitor, CPT (*), LPG (⧫), or LAMA (•). Note that YAHSP did solve all instances, though the poor quality of its solution prevents from displaying it for the sake of readibility.

[12] A. Gerevini, A. Saetti, and I. Serina. On Managing Temporal Information for Handling Durative Actions in LPG. In *AI*IA 2003: Advances in Artificial Intelligence*. Springer Verlag, 2003.

[13] A. Gerevini, A. Saetti, and I. Serina. Planning through Stochastic Local Search and Temporal Action Graphs in LPG. *JAIR*, 20:239–290, 2003.

[14] J. J. Grefenstette. Optimization of control parameters for genetic algorithms. *IEEE Trans. on Systems, Man and Cybernetics*, SMC-16, 1986.

[15] W. Hart, N. Krasnogor, and J. Smith, editors. *Recent Advances in Memetic Algorithms*. Studies in Fuzziness and Soft Computing, Vol. 166. Springer Verlag, 2005.

[16] P. Haslum and H. Geffner. Admissible Heuristics for Optimal Planning. In *Proc. AIPS-2000*, pages 70–82, 2000.

[17] F. Hutter, Y. Hamadi, H. H. Hoos, and K. Leyton-Brown. Performance prediction and automated tuning of randomized and parametric algorithms. In *CP 2006*, number 4204 in lncs, pages 213–228. Springer Verlag, 2006.

[18] F. Hutter, H. H. Hoos, K. Leyton-Brown, and K. P. Murphy. An experimental investigation of model-based parameter optimisation: SPO and beyond. In Franz Rothlauf et al., editor, *GECCO'09*, pages 271–278. ACM, 2009.

[19] F. Lobo, C. Lima, and Z. Michalewicz, editors. *Parameter Setting in Evolutionary Algorithms*. Springer, Berlin, 2007.

[20] F. Marcotorchino and P. Michaud. Agrégation des similarités en classification automatique. *In Revue de Statistique Applique*, Vol 30-No 2, 1981.

[21] O. Maron and A. W. Moore. Hoeffding Races: Accelerating Model Selection Search for Classification and Function Approximation. In *NIPS 6*, pages 59–66. Morgan Kaufmann, 1994.

[22] D. McDermott. PDDL – The Planning Domain Definition language. At `http://ftp.cs.yale.edu/pub/mcdermott`, 1998.

[23] I. Muslea. SINERGY: A Linear Planner Based on Genetic Programming. In *Proc. ECP '97*, pages 312–324. Springer-Verlag, 1997.

[24] V. Nannen and A. E. Eiben. Relevance estimation and value calibration of evolutionary algorithm parameters. In M. Veloso, editor, *Proc. Intl. Joint Conference on Artificial Intelligence*, pages 975–980, 2007.

[25] S. Richter, M. Helmert, and M. Westphal. Landmarks Revisited. In *AAAI'08*, pages 975–982. AAAI Press, 2008.

[26] M. Schoenauer, P. Savéant, and V. Vidal. Divide-and-Evolve: a New Memetic Scheme for Domain-Independent Temporal Planning. In J. Gottlieb and G. Raidl, editors, *Proc. EvoCOP'06*. Springer Verlag, 2006.

[27] M. Schoenauer, P. Savéant, and V. Vidal. Divide-and-Evolve: a Sequential Hybridization Strategy using Evolutionary Algorithms. In Z. Michalewicz and P. Siarry, editors, *Advances in Metaheuristics for Hard Optimization*, pages 179–198. Springer, 2007.

[28] L. Spector. Genetic Programming and AI Planning Systems. In *Proc. AAAI 94*, pages 1329–1334. AAAI/MIT Press, 1994.

[29] V. Vidal. A Lookahead Strategy for Heuristic Search Planning. In 14^{th} *International Conference on Automated Planning & Scheduling - ICAPS*, pages 150–160, 2004.

[30] V. Vidal and H. Geffner. Branching and Pruning: An Optimal Temporal POCL Planner based on Constraint Programming. *Artificial Intelligence*, 170(3):298–335, 2006.

[31] C. H. Westerberg and J. Levine. "GenPlan": Combining Genetic Programming and Planning. In M. Garagnani, editor, *19th PLANSIG Workshop*, 2000.

[32] C. H. Westerberg and J. Levine. Investigations of Different Seeding Strategies in a Genetic Planner. In E.J.W. Boers et al., editor, *Applications of Evolutionary Computing*, pages 505–514. LNCS 2037, Springer-Verlag, 2001.

[33] T. Yu, L. Davis, C. Baydar, and R. Roy, editors. *Evolutionary Computation in Practice*. Studies in Computational Intelligence 88, Springer Verlag, 2008.

[34] B. Yuan and M. Gallagher. Statistical Racing Techniques for Improved Empirical Evaluation of Evolutionary Algorithms. In *Parallel Problem Solving from Nature - PPSN VIII*, LNCS 3242, pages 172–181. Springer Verlag, 2004.

[35] B. Yuan and M. Gallagher. Combining Meta-EAs and Racing for Difficult EA Parameter Tuning Tasks. In *Parameter Setting in Evolutionary Algorithms*, pages 121–142. Springer-Verlag, 2007.

#	Global racing		Domain racing		Instance racing		Best
Gold	Min	med.	Min	med.	Min	med.	CPT
1	**28**	31	**28**	31	**28**	31	28
2	**19**	21	**19**	21	**19**	21	19
3	**18**	21	**18**	20	**18**	20	18
4	**24**	28	**24**	27	**24**	25	23
5	**20**	20	**20**	20	**20**	20	20
6	**24**	26	**24**	26	**24**	26	24
7	**24**	29	**24**	28	**24**	28	24
8	**25**	27	**25**	27	**25**	27	25
9	**26**	30	27	29	**26**	29	26
10	**17**	19	**17**	19	**17**	19	17
11	**29**	33	**29**	33	**29**	33	29
12	**25**	32	**25**	32	**25**	33	25
13	**27**	34	**27**	32	**27**	29	27
14	**27**	28	**27**	28	**27**	27	27
15	**26**	34	**26**	32	**26**	32	26
16	**24**	27	**24**	27	**24**	27	24
17	**34**	41	**34**	39	**34**	39	32
18	**22**	32	**22**	31	**22**	30	22
19	**35**	38	36	38	**35**	38	31
20	**30**	38	**30**	35	**30**	38	30
21	**33**	35	**33**	46	**33**	34	31
22	30	51	29	43	**28**	41	28
23	33	43	**32**	46	33	43	32
24	**39**	55	**39**	52	**39**	52	39
25	**41**	173	**41**	62	**41**	50	37
26	36	50	36	50	**35**	50	31
27	39	50	39	49	**38**	46	34
28	**25**	40	**25**	41	**25**	36	25
29	30	41	**29**	38	**29**	38	29
30	**33**	130	**33**	34	**33**	34	31
zeno	Min	med.	Min	med.	Min	med.	LPG
1	**173**	173	**173**	173	**173**	173	173
2	**592**	599	**592**	599	**592**	592	592
3	**280**	280	**280**	280	**280**	280	280
4	529	529	**522**	529	**522**	529	522
5	**400**	400	**400**	400	**400**	400	400
6	**323**	323	**323**	323	**323**	323	323
7	672	692	**665**	692	**665**	679	665
8	**522**	522	**522**	522	**522**	522	522
9	**522**	629	**522**	536	**522**	536	522
10	**453**	636	**453**	636	**453**	636	453
11	433	433	**423**	453	**423**	433	423
12	**549**	626	**549**	616	**549**	603	549
13	659	659	633	659	**626**	659	596
14	**503**	1009	633	905	**503**	633	519
15	756	962	782	962	**709**	962	736
16	906	1438	872	1432	**653**	1292	683
17	1658	2454	1462	2623	**1324**	2444	1189
18	1547	2304	1801	2300	**1152**	2114	1312
19	1968	2740	1700	2767	**1691**	2710	1698
20	1780	2900	**1628**	2860	**1628**	2860	1898
Open	Min	med.	Min	med.	Min	med.	LAMA
1	**2**	2	**2**	2	**2**	2	2
2	**3**	3	**3**	3	**3**	3	3
3	**2**	2	**2**	2	**2**	2	2
4	**2**	3	**2**	3	**2**	3	2
5	**2**	2	**2**	2	**2**	2	2
6	**4**	4	**4**	4	**4**	4	4
7	**4**	4	**4**	4	**4**	4	4
8	**4**	4	**4**	4	**4**	4	4
9	**4**	4	**4**	4	**4**	4	4
10	**4**	5	**4**	5	**4**	4	4
11	**5**	6	**5**	5	**5**	5	10
12	**3**	5	**3**	5	**3**	4	3
13	**5**	6	**5**	6	**5**	6	10
14	5	7	5	7	**4**	7	10
15	**5**	7	**5**	6	**5**	6	10
16	8	10	**7**	10	**7**	10	10
17	8	8	**7**	9	**7**	8	10
18	**6**	8	7	8	**6**	7	18
19	12	14	11	14	**10**	13	11
20	14	17	15	22	**12**	16	12
21	10	16	10	16	**9**	12	10
22	18	29	17	28	**16**	19	14
23	17	27	14	26	**13**	14	10
24	13	24	14	25	**12**	16	11
25	25	39	27	38	**21**	38	20
26	22	36	22	36	**18**	35	15
27	27	39	25	37	**22**	25	21
28	36	53	37	52	**33**	52	32
29	42	53	35	54	**33**	54	30
30	37	54	33	53	**33**	53	34

3. INSTANCE-BASED PARAMETER TUNING FOR EVOLUTIONARY AI PLANNING

The following paper presents the preliminary results of an original framework for instance-based off-line parameter tuning based on an empirical model of instance hardness that is learned from previous runs. Note that this framework could be used in other contexts than DaE. First examples of similar approach dates back to the work of Hutter et al. [13][1], for SAT problems: instances are described by some features, and extensive experiments are run on many instances with many different solvers (or many different parameterizations of some solvers), and the performance of each solver on each instance is recorded. Machine Learning is then used to learn the mapping from (instance,solver) to performance. When a new instance is to be handled, the best solver is determined from the learned mapping, i.e., the one that should give the best performance according to the learned mapping.
There are three main issues when attempting to use such method on a new type of problem.

- The design of the features that will be used to describe each instance of the problem. They should be discriminant enough so as to separate instances on which some algorithm behave differently. For the SAT domain [13], several dozens of features had been proposed in the literature, out of which 45 were chosen. No such features exist in the AI Planning area, and the preliminary work presented here is based on 14 elementary features gathered from statistics on the instance after the initial *grounding* step.
- The choice of the performance that will be the output of the predictive model. For SAT problems, because only satisfiable problems were considered in [13], a natural performance is the time to solution. However, in the AI Planning domain, there is in general no known solution - and in any case, many algorithms or parameter setting never reach this optimal solution. Also, it is very difficult to set some general target such as “reach the best makespan up to α%” because there is no possible normalization of the makespans across instances. So another point of view had to be adopted, and the performance is computed the way it has been proposed in the IPC competitions (see e.g., [16] for a detailed description). Such performance was designed to allow a sound aggregation of the results across different instances.
- The choice of the sample instances on which the model will be learned. This issue has already been discussed in the first paper presented in this document: the more instances the better, from a generalization point of view. But the more instances the higher the computational cost, and some trade-off had to be used.

Another issue regards the choice of the Machine Learning tool: Feed-forward Artificial Neural Networks with standard backpropagation algorithms are used here, though many other paradigms could have been used, and the results would probably have been similar.

However, there are two main novelties in the *Learn-and-Optimize* method proposed here:

- First, the kind of mapping that is learned is not the (instance, parameters) to performance mapping, but directly the mapping from instance to optimal parameters.
- Second, the learning and optimization phases are intertwined: whereas in [13] all runs were run at once, and learning was achived in a single pass over the whole set of examples, the learning and optimization steps are much more intertwined in the LaO approach.

The preliminary results presented in the following are promising, and on-going work is concerned with addressing the first issue above – the design of sound features – as well as validaring the proposed approach on larger sets of instances.

[1] The citations refer to the references of the following paper.

Instance-Based Parameter Tuning for Evolutionary AI Planning

This work is accepted as poster to the *Self-* Search* track of GECCO'2011

ABSTRACT

Learn-and-Optimize (LaO) is a generic surrogate based method for parameter tuning combining learning and optimization. In this paper LaO is used to tune Divide-and-Evolve (DaE), an Evolutionary Algorithm for AI Planning. The LaO framework makes it possible to learn the relation between some features describing a given instance and the optimal parameters for this instance, thus it enables to extrapolate this relation to unknown instances in the same domain. Moreover, the learned model is used as a surrogate-model to accelerate the search for the optimal parameters. It hence becomes possible to solve intra-domain and extra-domain generalization in a single framework. The proposed implementation of LaO uses an Artificial Neural Network for learning the mapping between features and optimal parameters, and the Covariance Matrix Adaptation Evolution Strategy for optimization. Results demonstrate that LaO is capable of improving the quality of the DaE results even with only a few iterations. The main limitation of the DaE case-study is the limited amount of meaningful features that are available to describe the instances. However, the learned model reaches almost the same performance on the test instances, which means that it is capable of generalization.

Categories and Subject Descriptors

I.2.6 [**Computing Methodologies**]: Artificial Intelligence Learning Parameter learning

General Terms

Theory

Keywords

parameter tuning, AI Planning, evolutionary algorithms

1. INTRODUCTION

Parameter tuning is basically a general optimization problem applied off-line to find the best parameters for complex algorithms, for example for Evolutionary Algorithms (EAs). Whereas the efficiency of EAs has been demonstrated on several application domains [29, 18], they usually need computationally expensive parameter tuning. Consequently, one is tempted to use either the default parameters of the framework he is using, or parameter values given in the literature for problems that are similar to his one.

Being a general optimization problem, there are as many parameter tuning algorithms as optimization techniques [7, 19]. However, several specialized methods have been proposed, and the most prominent today are Racing [5], REVAC [21], SPO [2], and ParamILS [14]. All these approaches face the same crucial generalization issue: can a parameter set that has been optimized for a given problem be successfully used to another one? The answer of course depends on the similarity of both problems. However, even in an optimization domain as precisely defined as AI Planning, there are very few results describing meaningful similarity measures between problem instances. Moreover, until now, sufficiently precise and accurate features have not been specified that would allow the user to accurately describe the problem, so that the optimal parameter-set could be learned from this feature-set, and carried on to other problems with similar description. To the best of our knowledge, no design of a general learning framework with some representative domains of AI planning has been proposed, and no general experiments has been carried out yet in this direction.

In the SAT domain, however, one work must be given as an example of what can be done along those lines. In [13], many relevant features have been gathered based on half a century of SAT-research, and hundreds of papers. Extensive parameter tuning on several thousands of instances has allowed the authors to learn, using function regression, a meaningful mapping between the features and the running-time of a given SAT solver with given parameters. Optimizing this model makes it possible to choose the optimal parameters for a given (unknown) instance. The present paper aims at generalizing this work made in AI planning, with one major difference: the target will be here to optimize the fitness value for a given runtime, and not the runtime to solution – as the optimal solution is generally not known for AI planning problems.

Unfortunately, until now, nobody has yet proposed a set of features for AI Planning problems in general, that would be sufficient to describe the characteristics of a problem, like was done in the SAT domain [13]. This paper makes a step toward a framework for parameter tuning applied generally for AI Planning and proposes a preliminary set of fea-

tures. The Learn-and-Optimize (LaO) framework consists of the combination of optimizing (i.e., parameter tuning) and learning, i.e., finding the mapping between features and best parameters. Furthermore, the results of learning will already be useful during further the optimization phases, using the learned model as in standard surrogate-model based techniques (see e.g., [1] for a Gaussian-process-based approach). LaO can of course be applied to any target optimization methodology that requires parameter tuning. In this paper, the target optimization technique is Evolutionary Algorithms (EA), more precisely the evolutionary AI planner called Divide-and-Evolve (DaE). However, DaE will be here considered as a black-box algorithm, without any modification for the purpose of this work than its original version described in [17].

The paper is organized as follows: AI Planning Problems and the classical YAHSP solver are briefly introduced in section 2. Section 3 describes the evolutionary Divide-and-Evolve algorithm. Section 4 introduces the original, top level parameter tuning method, Learn-and-Optimize. The case study presented in Section 5 applies LaO to DaE, following the rules of the International Planning Competition 2011 – Learning Track. Finally, conclusions are drawn and further directions of research are proposed in Section 6.

2. AI PLANNING

An Artificial Intelligence (AI) planning problem is defined by the triplet of an initial state, a goal state, and a set of possible actions. An action modifies the current state and can only be applied if certain conditions are met. A solution plan to a planning problem is an ordered list of actions, whose execution from the initial state achieves the goal state. The quality criterion of a plan depends on the type of available actions: in the simplest case (e.g. STRIPS domain), it is the number of actions; it may also be the total cost of the pan for actions with cost; and it is the total duration of the plan, aka *makespan*, for temporal problems with so called durative actions.

Domain-independent planners rely on the Planning Domain Definition Language PDDL2.1 [8]. The history of PDDL is closely related to the different editions of the International Planning Competitions (IPCs `http://ipc.icaps-conference.org/`), and the problems submitted to the participants, written in PDDL, are still the main benchmarks in AI Planning. The description of a planning problem consists of two separate parts usually placed in two different files: the generic domain on the one hand and a specific instance scenario on the other hand. The domain file specifies object types and predicates, which define possible states, and actions, which define possible state changes. The instance scenario declares the actual objects of interest, gives the initial state and provides a description of the goal. A state is described by a set of atomic formulae, or atoms. An atom is defined by a predicate followed by a list of object identifiers: (PREDICATE_NAME OBJ_1 ... OBJ_N).

The initial state is complete, whereas the goal might be a partial state. An action is composed of a set of preconditions and a set of effects, and applies to a list of variables given as arguments, and possibly a duration or a cost. Preconditions are logical constraints which apply domain predicates to the arguments and trigger the effects when they are satisfied. Effects enable state transitions by adding or removing atoms.

A solution plan to a planning problem is a consistent schedule of grounded actions whose execution in the initial state leads to a state that contains the goal state, i.e., where all atoms of the problem goal are true. A planning problem defined on domain D with initial state I and goal G will be denoted in the following as $\mathcal{P}_D(I, G)$.

3. DIVIDE-AND-EVOLVE

Early approaches to AI Planning using Evolutionary Algorithms directly handled possible solutions, i.e. possible plans: an individual is an ordered sequence of actions see [25, 20, 27, 28, 6]. However, as it is often the case in Evolutionary Combinatorial optimization, those direct encoding approaches have limited performance in comparison to the traditional AI planning approaches. Furthermore, hybridization with classical methods has been the way to success in many combinatorial domains, as witnessed by the fruitful emerging domain of memetic algorithms [11]. Along those lines, though relying on an original "memetization" principle, a novel hybridization of Evolutionary Algorithms (EAs) with AI Planning, termed Divide-and-Evolve (DaE) has been proposed [23, 24]. For a complete formal description, see [16].

The basic idea of DaE in order to solve a planning task $\mathcal{P}_D(I, G)$ is to find a sequence of states $S_1, \ldots, S_n$, and to use some embedded planner to solve the series of planning problems $\mathcal{P}_D(S_k, S_{k+1})$, for $k \in [0, n]$ (with the convention that $S_0 = I$ and $S_{n+1} = G$). The generation and optimization of the sequence of states $(S_i)_{i\in[1,n]}$ is driven by an evolutionary algorithm. The fitness (quality criterion) of a list of partial states $S_1, \ldots, S_n$ is computed by repeatedly calling the external 'embedded' planner to solve the sequence of problems $\mathcal{P}_D(S_k, S_{k+1})$, $\{k = 0, \ldots, n\}$. The concatenation of the corresponding plans (possibly with some compression step) is a solution of the initial problem. Any existing planner can be used as embedded planner, but since guaranty of optimality at all calls is not mandatory in order for DaE to obtain good quality results [16], a sub-optimal, but fast planner is used: YAHSP [26] is a lookahead strategy planning system for sub-optimal planning which uses the actions in the relaxed plan to compute reachable states in order to speed up the search process.

A state is a list of atoms built over the set of predicates and the set of object instances. However, searching the space of complete states would result in a rapid explosion of the size of the search space. Moreover, goals of planning problem need only be to defined as partial states. It thus seems more practical to search only sequences of partial states, and to limit the choice of possible atoms used within such partial states. However, this raises the issue of the choice of the atoms to be used to represent individuals, among all possible atoms. The result of the previous experiments on different domains of temporal planning tasks from the IPC benchmark series [3] demonstrates the need for a very careful choice of the atoms that are used to build the partial states. The method used to build the partial states is based on an estimation of the earliest time from which an atom can become true. Such estimation can be obtained by any admissible heuristic function (e.g h^1, h^2... [12]). The possible start times are then used in order to restrict the candidate atoms for each partial state. A partial state is built at a given time by randomly choosing among several atoms that are possibly true at this time. The sequence of states is then

built by preserving the estimated chronology between atoms (time consistency).
An individual in DaE is hence represented as a variable-length ordered time-consistent list of partial states, and each state is a variable-length list of atoms that are not pairwise mutex, as far as the initial grounding of all atoms can tell (exactly determining if two atoms are mutex amounts to solving a complete planning problem). Furthermore, all operators that manipulate the representation (see below) maintain the chronology between atoms and the approximate local consistency of a state, i.e. avoid pairwise mutexes.
One-point crossover is used, adapted to variable-length representation in that both crossover points are independently chosen, uniformly in both parents. Four different mutation operators have been designed, and once an individual has been chosen for mutation (according to a population-level mutation rate), the choice of which mutation to apply is made according to user-defined relative weights. Because an individual is a variable length list of states, and a state is a variable length list of atoms, the mutation operator can act at both levels: at the individual level by adding (addState) or removing (delState) a state; or at the state level by adding (addAtom) or removing (delAtom) some atoms in the given state. The list of DaE parameters that will be tuned in this paper is given in Table 2.

4. LEARN-AND-OPTIMIZE FOR PARAMETER TUNING

4.1 The General LaO Framework

As already mentioned, parameter tuning is actually a general global optimization problem, thus facing the routine issue of local optimality. But a further problem arises in parameter tuning, and this is the generality of the tuned parameters. Tuning only one instance has of course a sense if only that instance is to be solved. Parameters tuned for one instance however, may not be optimal for other instances, as [4] demonstrates. Furthermore, this paper also demonstrates that parameter tuning for several domains simultaneously is even more difficult, if at all possible.
Even when generalizing parameters learned on one instance to another instance of the same domain (intra-domain generalization) might be problematic, as there are instances with very different complexity in the same domain. For example in [4] per-domain tuning was performed with the most difficult, largest instance, considered as a representative of the whole domain. However, it is clear from the results that these parameters were often suboptimal for the other instances. And similar issue might arise even when using several instances as representatives of the domain. Since the optimum values of the parameters might change from instance to instance, only a "dull" average-like parameter-setting may be computed. Moreover, the computational cost of parameter tuning increases linearly with the number of training instances.
The issue is of course even more critical when aiming at inter-domain generalization, i.e., learning the parameters on one ir several instances, and using the learned parameters on instances of different domain than that of the training instances. Indeed, differences between the domains may cause a problem, and even instances of apparent similar complexity (e.g. same number of objects) may require different settings from domain to domain. The poor results with global tuning in [4] indicate that these are issues to be considered. One workaround this generalization issue is to relax the constraint of finding a single universally optimal parameter-set, that certainly does not exist, and to focus on learning a complex relation between instances and optimal parameters.
The propose Learn-and-Optimize framework (LaO) aims at learning such relation, thus, in the ideal case, solving both the intra-domain and extra-domain generalization problems, by adding learning to optimization. The underlying hypothesis is that there exists a relation between some features describing an instance and the optimal parameters for solving this instance, and the goal of this work is to propose a general methodology to do so. If well designed, the features should describe differences both between instances from the same domain, and differences between instances of different domains – and hence differences between domains, too. The case study analyzed here deals with AI planning, and some features extracted from both the domain-file and the instance-file will be proposed later.
Suppose for now that we have n features and m parameters, and we are doing per-instance parameter tuning on instance $\mathcal{I}$. For the sake of simplicity and generality, both the fitness, the features and the parameters are considered as real values. Parameter tuning is the optimization (e.g., minimization) of the fitness function $f_{\mathcal{I}} : \mathbf{R}^m \to \mathbf{R}$, the expected value of the stochastic algorithm DaE executed with parameter $p \in \mathbf{R}^m$. The optimal parameter set is defined by $p_{opt} = argmin_p\{f_{\mathcal{I}}(p)\}$.
For each instance $\mathcal{I}$, consider the set $F(\mathcal{I}) \in \mathbf{R}^n$ of the features describing this instance. Two relations have to be taken into account: each planning instance has features, and it has an optimal parameter-set. In order to be able to generalize, we have to get rid of the instance, and collapse both relations into one single relation between feature-space and parameter-space. This is only possible if different instances have different features, or, more generally, if instances having the same features also share the same optimal parameter set. Relaxing this tight constraint, we could simply assume that the features are such that if two instances have similar features, their optimal parameter sets is such that using either sets does not make a big difference regarding the optimization problem (i.e., for AI planning, leads to similar makespans). However, for the sake of simplicity let us assume that there exists an unambiguous mapping from the feature space to the optimal parameter space.

$$p_F : \mathbf{R}^n \to \mathbf{R}^m, p_F(F) = p_{opt} \quad (1)$$

.
However, we will indicate, if some problems in the results may be caused by an unambiguity. The relation p_F between features and optimal parameters can be learned by any supervised learning method capable of representing, interpolating and extrapolating $\mathbf{R}^n \to \mathbf{R}^m$ mappings, provided sufficient data are available.
A simple method could be to use any standard parameter tuning method for an appropriate training set of instances in a given domain, and then to use an appropriate supervised learning method in order to learn the relationship between the features and the best parameters. However, learning and optimizing may be combined, and this is the main idea behind LaO.

The idea of using some surrogate model in optimization is not new. Here, however, there are several instances to optimize, and only one model is available, that maps the feature-space into the parameter-space. Nevertheless, there is no question about how to use such a model of p_F in optimization: one can always ask the model for hints about a given parameter-set. Of course, if the model were perfectly fit to the training data, it would be useless, since it would return the same hint as trained. Therefore under-fitting when learning the mapping from feature-space to parameter-space is beneficial during the optimization phase in order to get new hints. One shall of course also avoid the regular threat on learning algorithms, that is over-fitting.
It seems reasonable that the stopping criterion of LaO is determined by the stopping criterion of the optimizer algorithm. After exiting one can also do a re-training of the learner with the best parameters found.
The proposed LaO algorithm is an open framework: one could use any appropriate learner for the mapping and any kind of optimizer for parameter tuning. LaO can of course be generalized to parameter tunning outside of AI planning. In most cases, where the parameters of an algorithm are to be tuned, there are instances of application, and in each of these cases, there is a possibility to improve the tuning by also learning the relation between some features and the optimal parameters.

4.2 An Implementation of LaO

A simple multilayer Feed-Forward Artificial Neural Network (ANN) trained with standard backpropagation was chosen here for the learning of the features-to-parameters mapping, though any other supervised-learning algorithm could have been used. The implicit hypothesis is that the relation p_F is not very complex, which means that a simple ANN may be used. In this work, one mapping is trained for each domain. Training a single domain-independent ANN is left for future work.
The other decision for LaO implementation is the choice of the optimizer used for parameter tuning. Because parameter optimization will be done successively for several instances, the simple yet robust (1+1)-Covariance Matrix Adaptation Evolution Strategy [10], in short (1+1)-CMA-ES, was chosen, and used with its robust own default parameters, as advocated in [4].
One original component, though, was added to some direct approach to parameter tuning: gene-transfer between instances. There will be one (1+1)-CMA-ES running for each instance, because using larger population sizes for a single instance would be far too costly. However, the (1+1)-CMA-ES algorithms running on all training instances form a population of individuals. The idea of Gene-Transfer is to use some crossover between the individuals of this population. Of course, the optimal parameter sets for the different instances are different; However, good 'chromosomes' for one instance may at least help another instance. Thus it may be used as a hint in the optimization of that other instance. Therefore random gene-transfer was used in the present implementation of LoA, by calling the so-called *Genetransferer*. When the Genetransferer is requested for a hint for one instance, it returns with uniform random distribution the so-far best parameter of a different instance (preventing, of course, that the default parameters are tried twice). Another benefit from gene-transfer is that it may smoothen out the ambiguities between instances, by increasing the probability for instances with the same features to test the same parameters, and thus the possibility to find out that the same parameters are appropriate for the same features.
Care must be taken when using the ANN and the Genetransferer as external hints within the standard CMA-ES process, to avoid corrupting it somewhat. Indeed, CMA-ES should be informed about these external hints, if they improve the fitness-function. The proposed solution is to handled those outcomers as if they were the hint of the CMA-ES algorithm, i.e. to replace a standard request from CMA-ES by the value of the external hint, thus minimizing possible corruption. The global step size is updated with true or false, depending on the improvement or lack of improvement, and as in the usual CMA-ES algorithm, the covariance matrix is updated only in the later case.
One additional technical difficulty arose with CMA-ES: each parameter is here restricted to an interval. This seems reasonable and makes the global algorithm more stable. Hence the parameters of the optimizer are actually normalized linearly onto the [0,1] interval. It is hence possible to apply a simple version of the box constraint handling technique described in [9], with a penalty term simply defined by $||p^{feas} - p||$, where p^{feas} is the closest value in the box. Moreover, only p^{feas} was recorded as a feasible solution , and later passed to the ANN. Note that the GeneTransferer and the ANN itself cannot send hints outside of the box. In order to not to compromise too much CMA-ES, several iterations of this were carried out for one hint of the ANN and one gene-transfer.
The implementation of LaO algorithm uses the Shark library [15] for CMA-ES and the FANN library for ANN [22]. To evaluate each parameter-setting with each instance, a cluster was used, that has approximately 60 nodes, most of them with 4 cores, some with 8. However, this cluster is used by many researchers, therefore our algorithm was automatically scheduled to only use the spare CPU cycles on this cluster. Because of the heterogeneity of the hardware architecture used here, it is not possible to rely on accurate predicted running times. Therefore, for each evaluation, the number of YAHSP evaluations is fixed for DaE. Note that the number of YAHSP evaluations is approximately proportional to the running time, so that the execution time for a particular computer is also determined independently of the parameter-settings. For example, even if the size of the population is increased, because of the fixed number of evaluations that is allowed, the number of generations will be limited accordingly in order to approximatively allow the same running time for each parameter-setting optimization. Moreover, since DaE is not deterministic, 11 independent runs were carried out for each DaE experiment with a given paramrter-set, and the fitness of this parameter set was taken to be the median fitness-value obtained by DaE.

5. RESULTS

In the Planning and Learning Part of IPC2011 (IPC), 5 sample domains were pre-published, with a corresponding problem-generator for each domain: Ferry, Freecell, Grid, Mprime, and Sokoban. Ferry and Sokoban were excluded from this study since there were not enough number of instances to learn any mapping. For each of the remaining 3 domains, 100 instances were generated, since this seemed to be appropriate for a running time of approximately 2-3

Domain Name	# of iterations	# training instances	# test instances	ANN error	quality-ratio in LaO	quality-ratio ANN on train	quality-ratio ANN on test
Freecell	16	108	230	0.1	1.09	1.05	1.04
Grid	10	55	124	0.09	1.09	1.05	1.03
Mprime	8	64	152	0.08	1.11	1.05	1.04

Table 1: Results by domains (only the actually usable training instances are shown). ANN-error is given as MSE, as returned by FANN. The quality-improvement ratio in Lao is that of the best parameter-set found by LaO.

Name	Min	Max	Default
Probability of crossover	0.0	1	0.8
Probability of mutation	0.0	1	0.2
Rate of mutation add station	0	10	1
, thus practically we can stop at some point. Rate of mutation delete station	0	10	3
Rate of mutation add atom	0	10	1
Rate of mutation delete atom	0	10	1
Mean average for mutations	0.0	1	0.8
Time interval radius	0	10	2
Maximum number of stations	5	50	20
Maximum number of nodes	100	100 000	10 000
Population size	10	300	100
Number of offspring	100	2 000	700

Table 2: DaE parameters that are controlled by LaO

Name	Default	CMA-ES	Transferer	ANN
Freecell	0 – 9	64 – 66	18 – 8	18 – 17
Grid	2 – 24	66 – 60	16 – 11	17 – 5
Mprime	2 – 45	59 – 36	21 – 11	18 – 8

Table 3: For each method (default, CMA-ES, Genetransferer or ANN), the percentage of instances on which this method gave the best parameter set. Each cell shows 2 figures: the first one considers all occurences of a method, no matter if another method also lead an equivalent parameter set, as good as the first one. The second figures only considers the first method (from left to right) that discovered the best parameter-set.

weeks: The competition track description fixes running time as 15 minutes. For each instance, 11 independent trials were run on a dedicated server to measure the median of number of evaluations with the default parameters. The termination criterion was the number of YAHSP evaluations. The median of those 11 runs was used as a termination criterion for each instance in the train set on any computer. However, many instances were never solved within 15 minutes, and those instances were dropped from the rest of experiment. The remaining instances were used for training.

Table 1 shows the data for each domain: from the 5 domaine, only 3 had enough solvable instances to be used for the learning part. The Mean Square Error (MSE) of the trained ANN is shown for each domain. But because the fitness takes only few values, there can be multiple optimal parameter sets for the same instance, resulting in an unavoidable MSE. One iteration of LaO amounts to 5 iterations of CMA-ES, followed by one ANN training and one Genetransferer. Due to the time constraints, only 10 iterations of LaO were run on the Grid, hence CMA-ES was called 50 times in total.

The ANN had 3 fully connected layers, and the hidden layer had the same number of neurons than the input. Standard back-propagation algorithm was used for learning (the default in FANN). In one iteration of LaO, the ANN was only trained for 50 iterations (aka epochs) without reseting the weights, in order to i-avoid over-training, and ii- making a gradual transition from the previous best parameter-set to the new best one, and eventually try some intermediate values. Hence. over the 10 iterations of LaO, 500 iterations (epochs) of the ANN were carried out in total. However, note that the best parameters were trained with much less iterations, depending on the time of their discovery. In the worst case, if the best parameter was found in the last iteration of LaO, it was trained for only 50 epochs and not used anymore. This explains why retraining is needed in the end.

A parameter-set in LaO may come from different sources, namely it can be the default parameter-set, or coming from CMA-ES, the Genetransferer, or as a result of applying the trained ANN to the instance features. Table 3 shows how each source contributes to the best overall parameter-settings. For each possible source, the first number is the ratio the source contributed to the best result if tie-breaks are taken into account, the second number shows the same, if only the first best parameter-set is taken into account. Note that the order of the sources is as it is in the table: for example if CMA-ES found a different parameter-settings with the same fitness than the default, this case that is not included in the first ratio, but is in the second. Analyzing both numbers leads to the following conclusions: for domain Mprime, the default parameter-settings was the optimal for 45% of the instances. However, only in 2% of the instances there was no other parameter-setting found with the same quality. In the domain Freecell, the share of ANN is quite high (18%), moreover we can see that in most cases, the other sources did not find a parameter-set with the same performance (17%). While Genetransferer in Freecell take equal share (18%) of all the best parameters, but only a part of them (8%) were unique. Note that CMA-ES was returning

the first hint in each iteration and had 5 times more possibilities than the ANN. Taking this into account, it is clear that both the ANN and Genetransferer made an important contribution to optimization.
LaO has been running for several weeks on a cluster. But this cluster was not dedicated to our experiments, i.e. only a small number of 4 or 8-core processors were available for each domain on average. After stopping LaO, retraining was made with 300 ANN epochs with the best data, because the ANN's saved directly from LaO may be under-trained. The MSE error of the ANN did not decrease using more epochs, which indicates that 300 iterations are enough at least for this amount of data and for this size of the ANN. Tests with 1000 iterations did not produce better results and neither training the ANN uniquely with the first found best parameters.
The controlled parameters of DaE are described in table 2. For a detailed description of these parameters, see [4]. The feature-set consists of 12 features. The first 5 features are computed from the domain file, after the initial grounding of YAHSP: number of fluents, goals, predicates, objects and types. One further feature we think could even be more important is called mutex-density, which is the number of mutexes divided by the number of all fluent-pairs. We also kept 6 less important features: number of lines, words and byte-count - obtained by the linux command "wc" - of the instance and the domain file. These features were kept only for historical reasons: they were used in the beginning as some "dummy" features.
Since testing was also carried out on the cluster, the termination criterion for testing was also the number of evaluations for each instance. For evaluation the quality-improvement the quality-ratio metric defined in IPC competitions was used. A baseline experiments comes from the default parameter-setting. The ratio of the fitness value for the default parameter and the tuned parameter was computed and average was taken over the instances in the train or test-set.

$$Q = \frac{Fitness_{baseline}}{Fitness_{tuned}} \quad (2)$$

Note that there was no unsolved instance in the training set, because they were dropped from the experiment if they were not solved with the default parameters. And indeed, it never happened that an instance that could be solved with the default parameters became unsolvable with an other set of parameters.
Table 1 presents several quality-improvement ratios. Label "in LaO" means that the best found parameter is compared to the default. By definition, this ratio can never be less than 1, because the default values are the starting point of the optimizations. Quality-improvement ratios for the retrained ANN on both the training-set and the test-set are also presented. In these later cases, numbers less then 1 are possible (the parameters resulting from the retrained ANN can give worse results that the ones given by the original ANN), but were rare. As can be seen on Table 1, some quality-gain in training was consistently achieved, but the transfer of this improvement to the ANN-model was only partial. The phenomenon can appear because of the unambiguity of the mapping, or because the ANN is complex enough for the mapping, or, and most probably, because the feature-set is not representative enough.
On the other hand, the ANN model generalizes excellently to the the independent test-set. Quality-improvement ratios dropped only by 0.01, i.e. the knowledge incorporated in the ANN was transferable to the test cases and usable almost to the same extent than for the train set.
The results are quite similar for all domain. Even the size of the training set seems not to be so crucial. For example for Freecell all the instances (108 out of 108 generated) could be used, because they were not so hard. On the other hand, only few Grid instances (55 out of 107 generated) could be used. However, both performed well. The explanation for this may be that both the 32 and 108 instances covered well the whole range of solvable instances.

6. CONCLUSIONS AND FUTURE WORK

The LaO method presented in this paper is a surrogate-model based combined learner and optimizer for parameter tuning. LaO was demonstrated to be capable of improving the quality of the DaE algorithm consistently, even though it was run only for a few iterations. On-going work is concerned with running LaO for an appropriate number of iterations. A clearly visible result is also that some of this quality-improvement can be incorporated into an ANN-model, which is also able to generalize excellently to an independent test-set.
Of course, LaO also has its own parameters, which should be tuned, too. Parameter tuning is in this respect similar to the question of the final parameters of an ultimate theory of the Universe: one can always try to reduce one theory to another one that possibly has less parameters. But such infinite regress can not be stopped: there will always be some ultimate parameters to be tuned. On the opposite, the parameters of any algorithm can be tuned by another algorithm and that may improve the results. However, there is no ultimate algorithm. The bad news here is that the computing capacity needed is exploding. The good news is that in each step in the hierarchy an improvement can be made. Nevertheless, the possible improvements become smaller and smaller.
The most important experiment to carry out in the future is simply to test the algorithm with more iterations and on more domains – and this will take several months of CPU even using a large cluster. Since LaO is only a framework, as indicated other kind of learning methods, and other kind of optimization techniques may be incorporated. If an ANN is used, the optimal structure has to be determined, or a more sophisticated solution is to apply one of the so-called Growing Neural Network architectures.
Also the benefit of gene-transfer and/or crossover should be investigated further. Gene-transfer shall be improved so that chromosomes are transfered deterministically, measuring the similarity of instances by the similarity of their features.
Finally, the inter-domain generalization capability of LaO must be tested: It might be possible to learn a mapping for many domains, since the features may grasp the specificity of a domain, and several domains might share some specificities. However, the present results indicate that the current feature set is too small and should be extended for better results. Feature-selection would then become important only if the number of features is large compared to the number of examples. Unfortunately, this is not the case yet.

7. ACKNOWLEDGEMENTS

This work is funded through French ANR project DESCARWIN ANR-09-COSI-002.

8. REFERENCES

[1] R. Bardenet and B. Kégl. Surrogating the surrogate: accelerating gaussian-process-based global optimization with a mixture cross-entropy algorithm. In *Proceedings of the 27th International Conference on Machine Learning (ICML 2010)*, 2010.

[2] T. Bartz-Beielstein, C. Lasarczyk, and M. Preuss. Sequential parameter optimization. In B. McKay, editor, *Proc. CEC'05*, pages 773–780. IEEE Press, 2005.

[3] J. Bibai, P. Savéant, and M. Schoenauer. Divide-And-Evolve Facing State-of-the-Art Temporal Planners during the 6^{th} International Planning Competition. In C. Cotta and P. Cowling, editors, *EvoCOP'09)*, number 5482 in LNCS, pages 133–144. Springer-Verlag, 2009.

[4] J. Bibai, P. Savéant, M. Schoenauer, and V. Vidal. On the generality of parameter tuning in evolutionary planning. In J. B. et al., editor, *Genetic and Evolutionary Computation Conference (GECCO)*, pages 241–248. ACM Press, July 2010.

[5] M. Birattari, T. Stützle, L. Paquete, and K. Varrentrapp. A Racing Algorithm for Configuring Metaheuristics. In *GECCO '02*, pages 11–18. Morgan Kaufmann, 2002.

[6] A. H. Brié and P. Morignot. Genetic Planning Using Variable Length Chromosomes. In *Proc. ICAPS*, 2005.

[7] A. E. Eiben, Z. Michalewicz, M. Schoenauer, and J. E. Smith. Parameter control in evolutionary algorithms. In Lipcoll et al. [18], chapter 2, pages 19–46.

[8] M. Fox and D. Long. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *JAIR*, 20:61–124, 2003.

[9] N. Hansen, S. Niederberger, L. Guzzella, and P. Koumoutsakos. A method for handling uncertainty in evolutionary optimization with an application to feedback control of combustion. *IEEE Transactions on Evolutionary Computation*, 13(1):180–197, 2009.

[10] N. Hansen and A. Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation*, 9(2):159–195, 2001.

[11] W. Hart, N. Krasnogor, and J. Smith, editors. *Recent Advances in Memetic Algorithms*. Studies in Fuzziness and Soft Computing, Vol. 166. Springer Verlag, 2005.

[12] P. Haslum and H. Geffner. Admissible Heuristics for Optimal Planning. In *Proc. AIPS-2000*, pages 70–82, 2000.

[13] F. Hutter, Y. Hamadi, H. H. Hoos, and K. Leyton-Brown. Performance prediction and automated tuning of randomized and parametric algorithms. In *CP 2006*, number 4204 in lncs, pages 213–228. Springer Verlag, 2006.

[14] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle. ParamILS: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306, October 2009.

[15] C. Igel, T. Glasmachers, and V. Heidrich-Meisner. Shark. *Journal of Machine Learning Research*, 9:993–996, 2008.

[16] Jacques Bibai, Pierre Savéant, Marc Schoenauer, and Vincent Vidal. An evolutionary metaheuristic based on state decomposition for domain-independent satisficing planning. In *ICAPS 2010*, pages 18–25. AAAI press, 2010.

[17] Jacques Bibai, Pierre Savéant, Marc Schoenauer, and Vincent Vidal. On the benefit of sub-optimality within the divide-and-evolve scheme. In P. Cowling and P. Merz, editors, *EvoCOP 2010*, number 6022 in Lecture Notes in Computer Science, pages 23–34. Springer-Verlag, 2010.

[18] F. Lobo, C. Lima, and Z. Michalewicz, editors. *Parameter Setting in Evolutionary Algorithms*. Springer, Berlin, 2007.

[19] E. Montero, M.-C. Riff, and B. Neveu. An evaluation of off-line calibration techniques for evolutionary algorithms. In *Proc. ACM-GECCO*, pages 299–300. ACM, 2010.

[20] I. Muslea. SINERGY: A Linear Planner Based on Genetic Programming. In *Proc. ECP '97*, pages 312–324. Springer-Verlag, 1997.

[21] V. Nannen, S. K. Smit, and A. E. Eiben. Costs and benefits of tuning parameters of evolutionary algorithms. In *Proceedings of the 20th Conference on Parallel Problem Solving from Nature*, 2008.

[22] N. Nissen. Implementation of a Fast Artificial Neural Network Library (FANN). Technical report, Department of Computer Science University of Copenhagen (DIKU), 2003.

[23] M. Schoenauer, P. Savéant, and V. Vidal. Divide-and-Evolve: a New Memetic Scheme for Domain-Independent Temporal Planning. In J. Gottlieb and G. Raidl, editors, *Proc. EvoCOP'06*. Springer Verlag, 2006.

[24] M. Schoenauer, P. Savéant, and V. Vidal. Divide-and-Evolve: a Sequential Hybridization Strategy using Evolutionary Algorithms. In Z. Michalewicz and P. Siarry, editors, *Advances in Metaheuristics for Hard Optimization*, pages 179–198. Springer, 2007.

[25] L. Spector. Genetic Programming and AI Planning Systems. In *Proc. AAAI 94*, pages 1329–1334. AAAI/MIT Press, 1994.

[26] V. Vidal. A lookahead strategy for heuristic search planning. In *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS'04)*, pages 150–159, Whistler, BC, Canada, June 2004. AAAI Press.

[27] C. H. Westerberg and J. Levine. "GenPlan": Combining Genetic Programming and Planning. In M. Garagnani, editor, *19th PLANSIG Workshop*, 2000.

[28] C. H. Westerberg and J. Levine. Investigations of Different Seeding Strategies in a Genetic Planner. In E.J.W. Boers et al., editor, *Applications of Evolutionary Computing*, pages 505–514. LNCS 2037, Springer-Verlag, 2001.

[29] T. Yu, L. Davis, C. Baydar, and R. Roy, editors. *Evolutionary Computation in Practice*. Studies in Computational Intelligence 88, Springer Verlag, 2008.