Adaptive Online Time Allocation to Search Algorithms

Matteo Gagliolo, Viktor Zhumatiy, and Jürgen Schmidhuber

IDSIA, Galleria 2, 6928 Manno-Lugano, Switzerland
{matteo,viktor,juergen}@idsia.ch

Abstract. Given is a search problem or a sequence of search problems, as well as a set of potentially useful search algorithms. We propose a general framework for online allocation of computation time to search algorithms based on experience with their performance so far. In an example instantiation, we use simple linear extrapolation of performance for allocating time to various simultaneously running genetic algorithms characterized by different parameter values. Despite the large number of searchers tested in parallel, on various tasks this rather general approach compares favorably to a more specialized state-of-the-art heuristic; in one case it is nearly two orders of magnitude faster.

1 Introduction

Suppose we have a finite or infinite set A of search algorithms $a_1, a_2, \ldots$. For example, A could be the infinite set of all programs of a particular programming language, or a finite set of genetic algorithms (GAs) [6] that differ only by certain parameters such as mutation rate and population size.

Given some problem or problem sequence, we would like to *automatically* – not manually as in much traditional work! – collect experience with various elements of A, in order to allocate more time to more "promising" $a_i \in A$, such that the total computational effort is small. Towards this end we introduce the following general framework.

The life of our Adaptive Online Time Allocator (AOTA) consists of steps $1, 2, \ldots$. At step k it tries to solve current problem $r(k)$ by allocating discrete computation time $t(k)$ to algorithm $a(k) \in A$. After its time has expired, $a(k)$ will pause and output a D-dimensional data vector $d(k) \in \mathbb{R}^D$ conveying information such as: did $a(k)$ converge on $r(k)$? How much did $a(k)$ improve some $r(k)$-specific fitness function? For all k, the pair

$$(a(k), t(k)) = f(r(k), H(k), P) \tag{1}$$

is a function of the historic experience set $H(k) = \{(i, r(i), a(i), t(i), d(i)) : 0 < i < k\}$ and the initial bias P, typically a probability distribution over A.

If the problems $r(k)$ change over time we may speak of *inter-problem* adaptation, or *inductive transfer* from one problem to the next. Otherwise we speak of *intra-problem* adaptation[1].

[1] Note that we always refer to the adaption of the time allocation schedule, not of the parameters themselves. The parameters of each a stay fixed. A broader class of AI algorithms, which we do not consider here, deals with the adaption of parameter values themselves, e.g. by decreasing a learning rate according to the current convergence, etc.

In the following we briefly present a few existent (sect. 1.1) and possible (sect. 1.2) examples of both kinds of adaptation, then we introduce our intra-problem approach applied to GA parameter selection (sect. 1.3), and give experimental results for it (sect. 2).

1.1 Previous Work

Certain previous methods may be viewed as instances of this framework. For example, the A of the Optimal Ordered Problem Solver (OOPS) [12] may include all programs of a universal programming language. The initial bias P is a probability distribution on A. As OOPS is solving more and more problems, it may reuse programs computing solutions to previous problems, increasing the probabilities of formerly unlikely programs in bias-optimal fashion [12], and with an optimal order of computational complexity. However, the feedback data $d(k)$ used by OOPS is quite limited: it just says whether $a(k)$ has halted or not, and whether it has solved the current problem. That is, OOPS performs inter-problem adaptation and does not exploit intra-problem information.

A theoretical description of a similar inter-problem adaption scheme, with some form of intra-problem adaption, is given in [14], where a system solves function inversion and time-limited optimization problems by searching in a space of problem solving techniques described in a Lisp-like language, allocating time to them according to their probabilities, and updating the probabilities according to positive and negative results on a sequence of problems.

Examples of simple intra-problem AOTAs are racing algorithms [10, 1]. Their finite A contains different parameterizations of a given supervised algorithm. Each is repeatedly run on a sequence of leave-one-out training sets; the $d(k)$ provide current mean errors and relative confidence intervals of each $a \in A$; badly performing $a \in A$ are progressively discarded as statistically sufficient evidence is gathered against them. This straight-forward approach, however, cannot be immediately extended to algorithms with unknown convergence time, e.g. GAs.

The "parameterless GA" [5] also may be viewed as a specialized intra-problem AOTA. Its A contains generational GAs a_i with population size 2^i (without mutation), which are generated and executed according to a fixed interleaving schedule that assigns runtime to $a_i \in A$ in proportion to a variable bias p_i, initialized by 2^{-i}. Once the population of a_i converges, or some a_j $(j > i)$ achieves higher average fitness, p_i becomes 0 and all $p_j(j > i)$ are doubled. This simple heuristic is motivated by the fact that, in the absence of mutation, convergence of a population is irreversible, and once the population of a_{i+1} reaches the average fitness of the population of a_i while being run half the time of a_i, then the population of a_i is probably drifting too slowly, so one may safely stop executing it, as it is unlikely that a smaller population will eventually produce a better solution than a larger one.

1.2 Choices for Learning Procedure *f*

Generally speaking, we might use all kinds of well-known learning algorithms to implement f. For example, we could use all or recent parts of the history $H(k)$ to train a neural network or a support vector machine to predict $d(k + 1)$ from inputs representing possible combinations of $a(k + 1)$ and $t(k + 1)$. The predictive model could then be used to find a pair $(a(k+1), t(k+1))$ that maximizes predicted success. Or, in a Reinforcement Learning (RL) setting [8], we may view each sequence $H(k)$ as a point in a huge, possibly continuous *state space*. The *action* at step k is the selection of pair $(a(k), t(k))$; the *reward* for solving a problem is inversely proportional to the total search time. Meta-learning techniques for algorithm recommendation [13, 9, 3], in which the relative performances of different $a \in A$ are predicted based on results on previous problems, could in principle be used as a base for inter-problem adaption.

Next we will present first results with a novel AOTA system that allocates time to GAs differing in more than just one parameter, and that implements f in a simple yet non-trivial way by extrapolating a linear best fit model of previous observations of performance improvements.

1.3 Intra-problem AOTA for Genetic Algorithms

The probability update scheme of the parameterless GA discussed above relies on specific issues related to population size. It cannot be extended to the choice of other parameters such as mutation rate. The AOTA system below is more general, but we will see that it does not suffer much from its increased generality, matching or even greatly exceeding the performance of the parameterless GA on several test problems.

We consider a single problem $r(k) = r$ for all k, a finite set of GAs $A = \{a_i, i = 1..n\}$ generated by combining different parameter settings, a bias $P(k) = \{p_i(k) = \Pr\{a_i\}$ at step $k, i = 1..n\}$ defined as a probability distribution over it, obtained by applying a function f_P to a set of values $U(k) = \{u_i(k), i = 1..n\}$ that we will relate to each algorithm's performance through a function f_u. Our $d(k)$ contains only the current average fitness of the individuals in the population of GA $a(k)$, that is, the history $H_i(k)$ of each a_i, $H_i(k) = \{(t(k), d(k)) | a(k) = a_i\}$, is simply a table of time versus average fitness values. Machine time is subdivided in small slots of duration ΔT, and a sequence of pairs $(a(k), t(k)) = (a_i, p_i \Delta T)$ is generated, allocating the slot to the various algorithms proportionally to this bias, as follows:

Method 1 (Bias based AOTA) *Initialize ΔT and $u_i(0)$, $i = 1..n$, and set $k = 1$.*

```
While (r not solved)
    Update $P_A(k) = f_P(U_A(k))$
    For each $i = 1..n$
        Generate and execute pair $(a(k), t(k)) = (a_i, p_i \Delta T)$
        Update $u_i(k+1) = f_u(H_i(k))$.
        $k = k + 1$
    End
End
```

As for the choice of f_u: from H_i, we try to give an estimate of the time $T_{i,sol}$ at which a_i would reach the target fitness value, and set $u_i(k) = 1/(T_{i,sol} - T_i(k))$, where $T_i(k) = \sum_{a(k)=a_i} t(k)$ is the time spent on a_i up to step k. In this way a suboptimal searcher on which we already have spent a lot of time can receive more credit than a faster searcher whose execution just started. With a linear regression based on a shifting window of the most recent c values in $H_i(k)$ (see Fig. 1), we estimate $T_{i,sol}$ as the time at which the resulting linear model predicts the target fitness value[2], rounded up to the nearest multiple of the time taken to run one generation of GA a_i, which is equivalent to population size z_i if we measure time in fitness function evaluations.

The size c of the shifting window used for the regression should be small enough, such that the linear model quickly reacts to changes in trend of the average fitness (e.g. when a population converges), but large enough to compensate for noise on the average fitness values. For example, different window sizes should be used for a_i's with different mutation rates, and preliminary experiments with a fixed window size gave best results for different values of c on different problems. For such reasons we plug *adaptive window sizing* into our AOTA framework, drawing inspiration from a financial forecasting method developed by Chow [4]. We set $c_i = 2$ and keep updating three linear models, one using window size c_i, the others $2 * c_i$, and $\max(2, c_i/2)$, respectively. At each step k we compare the new $d(k)$ with the values predicted by the three linear models. The window size for which the smallest error is achieved becomes the current c_i.

For all $a_i \in A$ the initial bias $p_i(0)$ is inversely proportional to population size z_i, as in parameterless GA. We initialize $u_i(0) = \text{const}/z_i$; each time an a_i is run for the first time, the estimated u_i is used to initialize $u_j = u_i z_i / z_j$ for all a_j's that have not started yet. Subsequent updates are only due to f_u, such that this initial bias can be quickly modified by experience.

For f_P we use plain normalization $p_i = u_i / \sum_{j \in I} u_j$; and compare to a greedy normalization, in which half of the current time slot is allocated to the current best a_i, the other half according to normalized u_i values.

Our method strives to be *algorithm independent*, i.e. we view our a_i's as black box searchers with unknown inner structure and properties. For example, we do not exploit the fact that in absence of mutation convergence is irreversible; neither do we try to infer relationships between the various a_i, e.g. by interpolating their performances across parameter space. Each u_i is updated independently; the only interaction between the a_i is due to their competition for runtime, mediated by the normalization process f_P. The need of algorithm specific knowledge is limited to the choice of the algorithm set A, the state information d, and the initial bias P.

2 Experiments

We tested our method on the following set of problems:

CNF3 SAT $m\ n$ is a conjunctive normal form (CNF) satisfiability problem, with n clauses, each being a logical OR of three variables chosen from a set of m boolean variables and their negations. The problem consists in finding an instantiation of the variables that verifies all clauses, and is termed *satisfiable* if it can be solved.

[2] The algorithm already solves the problem when the best fitness value reaches the target, but this value is usually noisier than the average fitness, and leads to less reliable predictions. Some upper limit can be used if no target value is known: with a higher target y^*, more credit is given to the slope α_i of the linear model, and less to the current average fitness y_i obtained by a_i, as it can be shown that $p_i \propto \frac{\tan \alpha_i}{y^* - y_i}$.

It can be expressed as a fitness maximization problem and addressed by a GA with bitstrings of length m as genomes, each representing an instantiation of the variables, with fitness equal to the number of satisfied clauses, between 0 and n. We chose three satisfiable instances from SATLIB [7], namely CNF3 SAT 20 91 (20 variables, 91 clauses, file `uf20-91/uf20-01.cnf`); CNF3 SAT 50 218 (50 variables, 218 clauses, file `uf50-218/uf50-01.cnf`); CNF3 SAT 75 325 (75 variables, 325 clauses, file `uf75-325/uf75-01.cnf`).

ONEMAX a simple toy problem, also reported in [5], that consists in maximizing the number of ones in a bitstring of length 100.

TRAP a deceptive artificial problem from [5], consisting in 10 copies of a 4-bit trap function. Each 4-bit block of a bitstring of length 40 gives a fitness contribution of 4 if all four bits are 1, and of $3 - q$ if $q < 4$ bits are 1.

TSP The Traveling Salesman Problem (TSP) requires to find the shortest path through a fully connected graph, without revisiting nodes. It can be solved by a GA with node lists as genomes. We expressed it as a fitness maximization problem, in which a path of length l is given fitness $1/l$. We considered two real instances from TSPLIB [11], with known best path: TSP burma 14 (14 nodes, file `burma14.tsp`) and TSP gr 17 (17 nodes, file `gr17.tsp`).

In all cases fitness values were normalized between 0 and 1. Machine time was always realistically measured in number of fitness evaluations, such that a GA with population size z takes z time steps to run for one generation.

In a first group of experiments we used the exact settings of the parameterless GA in [5]: a set A_1 of 19 simple generational GAs, with no mutation, uniform crossover with probability 0.5, tournament selection without replacement, tournament size 4, and population sizes $2^i, i = 1..19$. We then repeated the experiments with a set A_2 of twice the size (this implies double-sized search space), obtained by instantiating each algorithm in A_1 with mutation probabilities 0 and $0.7/L$, where L is the genome length, for a total of $19 \times 2 = 38$ competing algorithms. The search space was made again twice as large in set A_3 by adding a binary parameter representing a choice between uniform and one-point crossover ($19 \times 2 \times 2 = 76$ competing search algorithms).

Distance Preserving Crossover (DPX [2]) was used for TSP problems, along with swap mutation; Partial Match Crossover, as implemented in GALib [15], was added in A_3.

In table 1 we give results for four variations of our method, obtained by combining plain (p) and greedy (g) probability update with fixed (f) and adaptive (a) window size. We label these methods AOTA$_{pf}$, AOTA$_{pa}$, AOTA$_{gf}$, AOTA$_{ga}$.

For comparison, we also give results for a parameterless GA over set A_1 only, labeled PLESS[3]; and of the *a priori unknown* fastest (on average) element of the three sets, labeled bestGA. The latter provoked counter-intuitive results, especially on set A_1, where bestGA is much slower than the time allocation procedures. This is because GAs in set A_1 do not use mutation, so a very large population is required to avoid premature convergence on all 40 runs executed, even if in most cases a smaller population

[3] Since the set A of this algorithm is in principle unbounded, we applied a variation of the algorithm with limited maximal population size: to avoid penalizing PLESS, we ran the limit population whenever a larger population was requested.

Table 1. Mean time (upper part), expressed in fitness function evaluations, and overhead (lower part) for each problem (columns), with 0.95 confidence intervals. Rows correspond to different algorithms: four variations of our AOTA on three different algorithms sets; the Parameterless Genetic Algorithm PLESS [5]; the fastest (on average) element of the three sets, labeled BestGA. See Sect. 2 for details.

	CNF3 SAT 20 91	CNF3 SAT 50 218	CNF3 SAT 75 325	ONEMAX	TRAP	TSP burma14	TSP gr17
AOTA$_{pf}(A_1)$	6945 ± 2357	303060 ± 62335	399775 ± 172026	11755 ± 946	3963348 ± 874795	13486 ± 4258	28064 ± 7973
AOTA$_{pa}(A_1)$	6466 ± 2434	296863 ± 62227	413921 ± 191802	11013 ± 917	4020018 ± 900687	15360 ± 5288	28342 ± 7259
AOTA$_{gf}(A_1)$	6194 ± 1864	252444 ± 53584	363694 ± 161169	10216 ± 849	3376378 ± 790406	13131 ± 4129	27602 ± 7117
AOTA$_{ga}(A_1)$	6179 ± 2396	251212 ± 51853	361868 ± 175230	9443 ± 810	3426109 ± 744091	15333 ± 5176	27940 ± 7346
PLESS(A_1)	6632 ± 2644	340789 ± 81220	603822 ± 334786	10407 ± 849	3047100 ± 681355	27578 ± 9579	42335 ± 10787
best GA(A_1)	14029 ± 1450	251904 ± 11788	1305805 ± 52936	8320 ± 122	3135898 ± 138343	32768 ± 4110	39946 ± 4001
AOTA$_{pf}(A_2)$	3552 ± 986	193704 ± 52459	297488 ± 99135	7184 ± 659	5429640 ± 1396380	12027 ± 3963	26603 ± 7942
AOTA$_{pa}(A_2)$	5114 ± 1857	201848 ± 51284	334076 ± 94351	7809 ± 837	4883354 ± 1258636	11902 ± 4404	28001 ± 8389
AOTA$_{gf}(A_2)$	3210 ± 996	158633 ± 42683	245983 ± 90511	5509 ± 547	4913763 ± 1263639	11021 ± 3707	24263 ± 8781
AOTA$_{ga}(A_2)$	4998 ± 1370	165137 ± 39958	277641 ± 86052	5552 ± 584	4602811 ± 1163608	11974 ± 4333	21557 ± 5548
best GA(A_2)	901 ± 334	32043 ± 10656	23236 ± 5582	1317 ± 91	2385510 ± 259136	20275 ± 2219	28431 ± 3056
AOTA$_{pf}(A_3)$	4164 ± 1336	310671 ± 76962	407563 ± 110573	11754 ± 1020	88051 ± 12688	20624 ± 6940	77288 ± 24584
AOTA$_{pa}(A_3)$	5939 ± 1634	357713 ± 87217	499665 ± 118741	12203 ± 1340	78981 ± 11555	18628 ± 6657	46806 ± 16950
AOTA$_{gf}(A_3)$	4282 ± 1178	262784 ± 66468	331002 ± 84824	8502 ± 738	69698 ± 9628	20004 ± 7350	76455 ± 25324
AOTA$_{ga}(A_3)$	6133 ± 1574	273006 ± 71425	382091 ± 99094	8223 ± 932	61481 ± 9802	17697 ± 6216	40562 ± 11598
best GA(A_3)	901 ± 334	32043 ± 10656	23236 ± 5582	1317 ± 91	20480 ± 1139	20275 ± 2219	28431 ± 3056
AOTA$_{pf}(A_1)$	4.27 ± 0.50	3.57 ± 0.13	3.11 ± 0.09	2.93 ± 0.07	3.64 ± 0.15	3.90 ± 0.46	3.37 ± 0.44
AOTA$_{pa}(A_1)$	3.64 ± 0.26	3.48 ± 0.12	3.04 ± 0.10	2.73 ± 0.05	3.42 ± 0.15	3.92 ± 0.36	4.11 ± 0.46
AOTA$_{gf}(A_1)$	3.97 ± 0.61	2.97 ± 0.13	2.72 ± 0.09	2.54 ± 0.06	2.99 ± 0.17	3.67 ± 0.44	3.22 ± 0.38
AOTA$_{ga}(A_1)$	3.39 ± 0.37	2.97 ± 0.13	2.62 ± 0.10	2.34 ± 0.05	3.05 ± 0.17	3.81 ± 0.46	4.03 ± 0.58
PLESS(A_1)	3.47 ± 0.62	3.79 ± 0.27	3.74 ± 0.30	2.59 ± 0.04	2.79 ± 0.11	6.42 ± 0.67	6.00 ± 1.08
AOTA$_{pf}(A_2)$	8.36 ± 0.96	7.34 ± 0.44	6.36 ± 0.27	6.01 ± 0.25	8.32 ± 0.66	8.84 ± 0.90	6.91 ± 1.02
AOTA$_{pa}(A_2)$	6.85 ± 0.46	6.64 ± 0.29	6.06 ± 0.21	5.29 ± 0.26	7.49 ± 0.58	7.50 ± 0.47	8.02 ± 0.86
AOTA$_{gf}(A_2)$	7.33 ± 1.35	5.94 ± 0.43	5.24 ± 0.33	4.48 ± 0.30	7.43 ± 1.29	7.51 ± 0.86	6.18 ± 0.75
AOTA$_{ga}(A_2)$	7.67 ± 2.27	5.57 ± 0.42	4.99 ± 0.37	3.70 ± 0.20	6.55 ± 0.79	7.28 ± 0.61	7.06 ± 0.77
AOTA$_{pf}(A_3)$	13.15 ± 1.34	14.26 ± 0.81	12.34 ± 0.47	10.52 ± 0.44	12.02 ± 0.47	14.72 ± 1.36	13.92 ± 1.37
AOTA$_{pa}(A_3)$	12.99 ± 0.88	13.23 ± 0.59	11.66 ± 0.41	9.20 ± 0.36	10.88 ± 0.44	11.98 ± 0.72	13.26 ± 1.48
AOTA$_{gf}(A_3)$	12.39 ± 2.05	12.14 ± 1.05	10.48 ± 1.17	7.13 ± 0.49	9.31 ± 0.45	12.06 ± 1.01	13.63 ± 1.21
AOTA$_{ga}(A_3)$	12.38 ± 1.30	10.57 ± 0.82	9.20 ± 0.58	5.85 ± 0.43	8.37 ± 0.60	10.93 ± 1.12	12.86 ± 1.42

would suffice. For PLESS and AOTA, many GAs are available at each run: if one a_i reaches the solution on all but a few unlucky runs, in which a_{i+1} wins, the resulting average performance will be largely determined by the faster a_i.

A better performance indicator are *overhead values*: the ratios between total search time and time spent on the a_i that first solves the problem.

In most cases, our method performs at least as well as PLESS. It is remarkable that despite its large search space of parameter combinations and simultaneously running search algorithms, our rather generic algorithm achieves results comparable to those of a highly specific algorithm incorporating detailed knowledge of GAs.

As expected, moving from set A_1 to set A_2 generally improved the performance, as it added faster solvers to the set[4], except for the deceptive problem TRAP, in which mutation is actually a slight disadvantage. Here the size of A was doubled through adding a_i's with performances very similar to those of A_1; hence the total search time increased. Use of the largest (least biased) A_3 led to slightly worse but still comparatively good performances on TSP and the two more difficult SAT problems, while the improvement on TRAP was dramatic, corresponding to a speed-up factor of roughly 30: this problem can in fact be solved much faster using one-point mutation, as it tends to preserve its building blocks.

Concerning f_P, the greedy approach performs almost always better than plain normalization, but the latter might generally be less risky. Fixed and adaptive window size led to similar results, with a slight advantage for the first. But since the adaptive scheme relieves the user from setting one parameter, it is preferable.

Figure 1 displays snapshots from three instants in the solution of the ONEMAX problem on set A_3. A value τ_{22} is obtained as current time-to-goal estimate, subtracting the time already spent (T_{22}) from the time $T_{22,sol}$ at which the resulting linear model reaches the target fitness value (Fig. 1 a); u_{22} is then set to $1/\tau_{22}$, and a portion $p_{22} \propto u_{22}$ of next time slot ΔT is allocated to a_{22}. (Fig. 1 b) displays bar graphs relative to the state of the $a \in A_3$: probability values (P), total time spent (T), average (avg) and best fitness. The arrows point to data relative to a_{22}, whose parameters are: pop size 2^3, mutation rate 0.007, uniform crossover, and which will eventually be the first to solve the problem, reaching the target fitness value 1 at $T_{22} = 1072$, after a total time of 7268 fitness function evaluations.

3 Conclusion and Outlook

The literature on search and optimization algorithms includes many papers on methods whose performance depends on numerous parameters. Typically, these parameters are adjusted by hand. The time that went into fine-tuning them is usually not reported.

Here we made first steps towards establishing a general framework for *automatically* adjusting such parameters. We let numerous search algorithms with alternative parameter settings compete against each other, allocating more computation time to the "more promising" ones in online fashion, where the notion of "more promising" is derived from an adaptive model of the experience so far.

[4] Smaller populations, and shorter execution times, are typically needed in presence of mutation, at least when the fitness landscape is not deceptive.

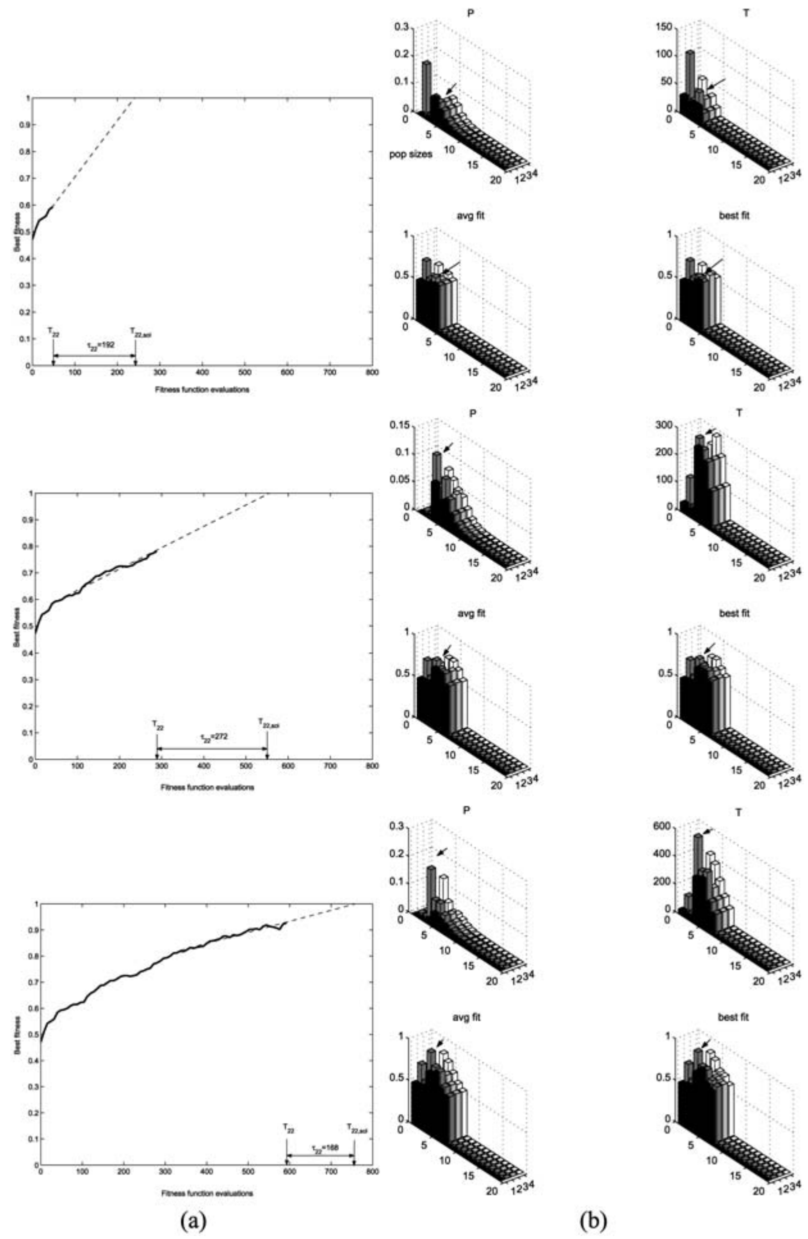

Fig. 1. Three snapshots from our AOTA at work, solving the ONEMAX problem on set A_3. (a) The learning curve for algorithm a_{22} (thick line), with its linear regression (thin dashed line). (b) Bar graphs relative to the state of the $a \in A_3$: probability values (P), total time spent (T), average (avg) and best fitness. The bars are lined according to population size z, from 1 ($z_1 = 2$) to 19 ($z_{19} = 2^{19}$): for each size there are four a differing in mutation rate (rows 1,3: 0; rows 2,4: 0.007) and crossover operator used (rows 1,2: uniform; rows 3,4: one-point). The arrows point to data relative to a_{22}, which will eventually be the first to solve the problem.

In a particular instantiation of this framework, we used a simple linear best fit model that maps online experience with algorithm-specific time allocation to expected performance improvements. The model is used to predict useful pairs of particular genetic algorithms and corresponding time allocations. Despite the relative generality of the approach and its comparatively large search space, its performance in terms of total computation time sometimes not only matches but greatly exceeds the one of a well-known but less general approach, although the latter has a much smaller search space motivated by human insight.

Future research will follow different directions, including the design of alternative adaptive performance models f_u, and their application to search spaces including algorithms other than GAs. The obtained techniques could then be used as plugins for inter-problem search, which is our longer-term goal.

A further step could be made leaving the framework presented in 1, in favour of a more general intra/inter-problem search in spaces of programs that can combine other algorithms as primitive actions, modifying their parameters online, possibly also performing conditional jumps to earlier parts of the code, in the style of the universal language used by OOPS [12].

References

1. M. Birattari, T. Stützle, L. Paquete, and K. Varrentrapp. A racing algorithm for configuring metaheuristics. In W. Langdon et al., editor, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 11–18. Morgan Kaufmann Publishers, 2002.
2. B. Freisleben and P. Merz. New Genetic Local Search Operators for the Traveling Salesman Problem. In H.-M. Voigt, W. Ebeling, I. Rechenberg, and H.-P. Schwefel, editors, *Parallel Problem Solving from Nature IV*, volume 1141 of *Lecture Notes in Computer Science*, pages 890–900. Springer, 1996.
3. J. Fürnkranz, J. Petrak, P. Brazdil, and C. Soares. On the use of fast subsampling estimates for algorithm recommendation. Technical Report TR-2002-36, Österreichisches Forschungsinstitut für Artificial Intelligence, Wien, 2002.
4. E. S. Gardner Jr. Exponential smoothing: the state of the art. *Journal of Forecasting*, 4:1–28, 1985.
5. G. R. Harick and F. G. Lobo. A parameter-less genetic algorithm. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, page 1867, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.
6. J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, 1975.
7. H. H. Hoos and T. Stützle. SATLIB: An Online Resource for Research on SAT. In T. Walsh I.P.Gent, H.v.Maaren, editor, *SAT 2000*, pages 283–292. IOS press, 2000. `http://www.satlib.org`.
8. L.P. Kaelbling, M.L. Littman, and A.W. Moore. Reinforcement learning: a survey. *Journal of AI research*, 4:237–285, 1996.
9. Alexandros Kalousis, Jo ao Gama, and Melanie Hilario. On data and algorithms: Understanding inductive performance. *Mach. Learn.*, 54(3):275–312, 2004.
10. A. W. Moore and M. S. Lee. Efficient algorithms for minimizing cross validation error. In *Proceedings of the 11th International Conference on Machine Learning*, pages 190–198. Morgan Kaufmann, 1994.

11. G. Reinelt. Tsplib 95. `http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95`.
12. J. Schmidhuber. Optimal ordered problem solver. *Machine Learning*, 54:211–254, 2004. Short version in *NIPS 15*, p. 1571–1578, 2003.
13. Carlos Soares, Pavel B. Brazdil, and Petr Kuba. A meta-learning method to select the kernel width in support vector regression. *Mach. Learn.*, 54(3):195–209, 2004.
14. Ray J. Solomonoff. Progress in incremental machine learning. Technical Report IDSIA-16-03, IDSIA, 2003.
15. M. Wall. GAlib, A C++ Genetic Algorithms Library. `http://lancet.mit.edu/ga`.