Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence (IJCAI-16)

Towards a White Box Approach to Automated Algorithm Design

Steven Adriaensen, Ann Nowé
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Elsene, Belgium
{steven.adriaensen, ann.nowe}@vub.ac.be

Abstract

To date, algorithms for real-world problems are most commonly designed following a manual, ad-hoc, trial & error approach, making algorithm design a tedious, time-consuming and costly process. Recently, Programming by Optimization (PbO) has been proposed as an alternative design paradigm in which algorithmic choices are left open by design and algorithm configuration methods (e.g. ParamILS) are used to automatically generate the best algorithm for a specific use-case. We argue that, while powerful, contemporary configurators limit themselves by abstracting information that can otherwise be exploited to speed up the optimization process as well as improve the quality of the resulting design. In this work, we propose an alternative white box approach, reformulating the algorithm design problem as a Markov Decision Process, capturing the intrinsic relationships between design decisions and their respective contribution to overall algorithm performance. Subsequently, we discuss and illustrate the benefits of this formulation experimentally.

1 Introduction

There are many ways to solve a given problem. Therefore, when faced with a problem, we must choose an algorithm to solve it (**the algorithm selection problem** [Rice, 1976]). In general, the best algorithm depends on the context in which it is used (e.g. problem instance, execution environment, etc.). Furthermore, these methods often have many parameters/components that require appropriate tuning and configuration to achieve satisfactory performance (**the algorithm configuration problem** [Hutter *et al.*, 2009]). To date, algorithms for many real-world problems are most commonly designed using a manual, ad-hoc, trial & error approach, making algorithm design a tedious, time-consuming and costly process, often leading to mediocre results.

Recently, Programming by Optimization (PbO) [Hoos, 2012] has been proposed as an alternative design paradigm in which difficult choices are deliberately left open at design time, resulting in a rich and potentially large design space, rather than a single algorithm. Subsequently, optimization methods are applied to automatically generate the best algorithm instance for a specific use-case. PbO distinguishes itself from Genetic Programming [Koza, 1992] in that it attempts to maximally utilize expert knowledge to prune the design space, which typically only consists of algorithms solving the problem considered correctly. To date, optimizers of choice for PbO are algorithm configuration methods. Algorithm configurators search a space of configurations to find one that optimizes the performance of an algorithm over a given set of training instances (inputs). Configurators such as ParamILS [Hutter *et al.*, 2009], iRace [López-Ibánez *et al.*, 2011] and GGA [Ansótegui *et al.*, 2009] are able to configure hundreds of parameters and represent the state-of-the-art. These methods require very little expert knowledge to use them, however, they are limited by the fact that they return a single configuration to be used on any input encountered in practice. As such, they don't solve the algorithm selection problem, making them susceptible to overfitting and *no-free-lunch* theorems [Wolpert and Macready, 1997]. Therefore, context-aware configurators have been proposed that solve both algorithm configuration and selection problems simultaneously by learning a configuration selection portfolio (e.g. Hydra [Xu *et al.*, 2010] and ISAC [Kadioglu *et al.*, 2010]).

In this paper, we address another weakness of contemporary algorithm configurators w.r.t. solving the Algorithm Design Problem (ADP): They consider algorithm performance as a **black box**, where a cost function maps a configuration and input to a real cost-value. They do not exploit the fact that this mapping is a consequence of algorithm execution, i.e. design decisions affect the execution path for an input in a particular way, which in turn relates to execution cost. To this purpose, we propose an alternative **white box** approach to the ADP. In Section 2 we reformulate the ADP as a Markov Decision Problem, capturing the intrinsic relationships between design decisions and their contribution to overall algorithm performance, including an explicit notion of context. Subsequently, we discuss possible solution approaches in Section 3. In Section 4 we discuss how white box information could be used to speed up the meta-optimization process and improve the quality of the resulting design. In Section 5 we describe a concrete implementation of a white box optimizer as a Proof of Concept, and use it to illustrate some of these benefits experimentally in Section 6. Finally, we discuss related research in Section 7 and conclude in Section 8.

2 The Algorithm Design Problem (ADP)

2.1 Black Box Formulation

Let Δ be the set of design choices, κ a function mapping each design choice to a set of alternative decisions and $C = \{c | c(dc) \in \kappa(dc), \forall dc \in \Delta\}$ the set of possible configurations. The traditional Black Box Algorithm Design Problem can be formulated as follows: Given a probability distribution $\mathcal{D}$ over the set of possible inputs X and an algorithm evaluation function $f : X \times C \to \mathbb{R}$, find a configuration c^* satisfying $c^* = \arg\max_{c'} \mathbb{E}_{x \sim \mathcal{D}} f(x, c')$, i.e. maximizing the expected evaluation function value w.r.t. $\mathcal{D}$.

2.2 White Box Formulation

In this section, we propose an alternative (white box) formulation, exposing the internal structure of f. Here, we consider algorithm design as a sequential decision process. Informally, we start executing an algorithm with open design choices. As long as the next instruction does not depend on the decision made for any of these, we simply continue execution. When it does, a choice point is reached, and a design decision must be made in order to continue execution. This process continues until termination. In what follows, we formalize this process. We first extend the Turing Machine (TM) [Turing, 1936] to include open design choices, their respective domains, and a notion of the desirability of an execution. Starting from the definition of a TM as a 7-tuple $\langle Q, \Gamma, B, \Sigma, \delta, q_0, F \rangle$ given in [Hopcroft, 1979, p. 319], we define an Algorithm Design Process as a 10-tuple: $TM_{adp} = \langle Q, \Gamma, B, \Sigma, \Delta, \kappa, \delta', \rho, q_0, F \rangle$:

Q is a finite, non-empty set of states.

Γ is a finite, non-empty set of tape alphabet symbols.

B is the blank symbol.

$\Sigma \subseteq \Gamma \setminus \{B\}$ is the set of input symbols.

Δ is the set of choice points or design choices.

κ is the domain function $\Delta \to 2^{Q \times \Gamma \times \{right, left\}}$.

δ' is the *open* transition function
$(Q \setminus F) \times \Gamma \to (Q \times \Gamma \times \{right, left\}) \cup \Delta$.

ρ is the reward function $(Q \setminus F) \times \Gamma \to \mathbb{R}$. Representing the contribution of a transition to overall performance (f).

$q_0 \in Q$ is the start state.

$F \subseteq Q$ is the set of *final* states.

Let $\alpha z b \beta$ be an Instantaneous Description (ID) of TM_{adp}, with current state or choice point z, tape content $\alpha b \beta$, head pointing at b. Here, α and β denote a prefix and suffix of zero, one or more tape symbols. Let $\alpha z \beta \vdash id'$ represent a single move of TM_{adp}. If $z \in \Delta$, an *agent* selects the next move $a \in \kappa(z)$ the TM will perform (denoted by $\vdash^a$), otherwise TM_{adp} is simulated as an ordinary TM. Finally, $id \vdash^* id'$ indicates that simulating TM_{adp} from id for zero, one or more moves results in id' for some choices of the agent.

Note that TM_{adp} can be seen as a Non-deterministic TM having rewards associated with its transitions. The objective of an agent in the ADP is to determine these non-deterministic transitions as to maximize the reward it accumulates: $\sum_{i=1}^{n} \rho(q_i, b_i)$ where q_i and b_i are the i^{th} of the n states and tape symbols read by TM_{adp}, before halting. In what follows we assume any simulation of TM_{adp} will eventually halt (independent of the agent's decisions).

We now rephrase TM_{adp} as a deterministic Markov Decision Process (MDP) [Bellman, 1957]: $MDP_{adp} = \langle S, A, T, R \rangle$:

$S = \{\alpha z \beta \mid \exists x \in X : q_0 x \vdash^* \alpha z \beta, z \in (\{q_0\} \cup \Delta)\} \cup \{\tau\}$ is the set of states, where τ is the terminal state.

$A = \bigcup_{s \in S} A_s$ is the set of actions, where $A_{\alpha z \beta} = \kappa(z)$ if $z \in \Delta$, $A_s = \{a_{noop}\}$ otherwise.

$T : S \times A_s \to S$ is the transition function. Let $next(id)$ be the ID resulting from simulating TM_{adp} starting from id until it halts or a choice point is reached.

$T(q_0 x, a_{noop}) = s'$. Let $\alpha z \beta = next(q_0 x)$.
If $z \in F$: $s' = \tau$, otherwise $s' = \alpha z \beta$.

$T(s, a) = s'$. Let $s \vdash^a id'$ and $\alpha z \beta = next(id')$.
If $z \in F$: $s' = \tau$, otherwise $s' = \alpha z \beta$.

$T(\tau, a_{noop}) = \tau$.

$R : S \times A_s \to \mathbb{R}$ is the reward function.

$R(s, a) = \sum_{i=1}^{n} \rho(q_i, b_i)$ where q_i and b_i are the i^{th} of the n states and tape symbols read by TM_{adp} while performing the simulation for $T(s, a)$.

$R(\tau, a_{noop}) = 0$

In state $\alpha z \beta$ the agent is faced with design choice z. We will refer to the combination of tape content ($\alpha \beta$) and head position as the *context* in which this design choice is to be made. A solution to an MDP is a policy $\pi : S \to A$, corresponding to an execution context dependent configuration. For a given policy, an MDP reduces to a Markov Chain and the formulation for TM_{adp} can be reduced to that of an ordinary Turing Machine, i.e. the algorithm corresponding to this configuration. The white box ADP as such reduces to finding a policy π^* satisfying

$$\pi^*(s) = \arg\max_{a \in A} R(s, a) + V_{\pi^*}(T(s, a)) \quad (1)$$

$$V_\pi(s) = R(s, \pi(s)) + V_\pi(T(s, \pi(s))) \qquad V_\pi(\tau) = 0$$

V_π is called the value function and gives the total reward that will be received following a given policy π from state s. In particular $V_\pi(q_0 x) = f(x, \pi), \forall x \in X$. Since $\pi*$ optimizes V for any state, it obtains optimal performance on **every** input and therefore **any** distribution $\mathcal{D}$.

Remark that MDP_{adp} is deterministic. Algorithms are never truly non-deterministic. Rather, stochasticity is a consequence of *pseudo-random* numbers computed deterministically from a given seed value, which is part of the input. Alternatively, assuming the seed value is drawn from a stationary distribution, we can reformulate transitions in MDP_{adp} to be stochastic: replacing $T(s, a)$ by $P(s, a, s')$, yielding the probability that taking action a in state s leads to state s'.

3 Solving the Algorithm Design Problem

Assuming S is finite, equation 1 (and its stochastic variant) can be solved using dynamic programming. Commonly used methods include Value and Policy iteration [Bellman, 1957; Howard, 1960], and their extensions. Note that these approaches require P and R to be known explicitly. However,

in our formulation, P and R are consequences of execution (TM simulation). In some cases, P and R can be derived from the source code. Otherwise, we can estimate P and R based on repeated simulation and compute π^* afterwards using dynamic programming. Alternatively, model-free Reinforcement Learning (RL) techniques can be used to learn π^* online, directly from the rewards and transitions observed during simulation [Sutton and Barto, 1998].

A complicating factor in solving real-world white box ADPs in practice is the fact that the context space (i.e. all possible memory states of the program) can be extremely large. Fortunately, the best decision will typically depend on only a subset of all context information. Furthermore, it is possible to exploit an arbitrary subset of observable/relevant features [Kaelbling *et al.*, 1998]. In addition, we can focus on relevant states, i.e. those that actually occur in practice (e.g. using prioritized sweeping, on-policy RL). Finally, solving MDPs with large state-action spaces has been an active research area for several decades [Wiering and van Otterlo, 2012], resulting in various techniques (e.g. incorporating function approximation). Exploring these in context of the implementation of a concrete solver is subject of future research.

4 Benefits of a White Box Approach

In this section we discuss potential benefits of following a white box approach to the ADP. First we briefly clarify the relationship between the white box ADP formulated in Section 2.2 and the traditional black-box ADP described in Section 2.1. These are not equivalent, rather the latter can be seen as a formulation of the former as a problem that can be solved using existing configurators. This formulation abstracts information that is naturally available in a PbO setting and restricts candidate designs to the set of static configurations. In Section 4.1 we describe how simulation-based solution techniques can exploit white box information to solve a given ADP faster. In Section 4.2 we discuss the benefit of considering adaptive designs as candidates in the ADP.

4.1 Speed up Meta-optimization

Black box evaluation of a configuration results in a single value (f), obtained after execution. White box evaluation on the other hand gives us a list of tuples (s_i, a_i, r_i), in chronological order, over the duration of the run, i.e. the state (choice point, context) s_i encountered, the action (decision) a_i taken, and the corresponding reward r_i received. Note that f can be deduced from this data ($f = \sum r_i$).

Black box evaluation of a configuration on one input does not provide any information about the performance of other configurations, or performance on other inputs. Since runs using different configurations and inputs might have large parts of their execution in common, i.e. encounter similar states, white box information can be generalized across designs and inputs, improving data efficiency. This fact can be exploited to perform more informed sampling and reduce the estimation error in a stochastic setting, i.e. speed up the optimization process. E.g. Design decisions only affect execution after they are made, i.e. only rewards received *after* making a certain decision can be attributed it. As a consequence, data can be shared across configurations for as far as they make the same decision in the states encountered thus far. Also, often only a subset of choice points is encountered during a run, i.e. multiple configurations correspond to the same execution path. In Section 5.2 we describe PURS, an agent using this information to sample uniformly at random w.r.t. execution paths, rather than configurations. Remark that some black-box optimizers allow the user to specify dependencies between design choices. However, this is only possible if these are *input independent* and demands considerable expert knowledge. Also, methods attempting to profit from relations between design decisions do exist (e.g. SMAC [Hutter *et al.*, 2011]). However, they do so by analyzing the cost function as a black-box, rather than by exploiting its internal structure.

Finally, black-box optimization typically requires many evaluations, which is problematic when these are expensive. Since white box information becomes available at runtime and states often repeat themselves within a single run, online learning (e.g. RL) can be used to evaluate multiple configurations over the course of a single execution. Which is for example useful to optimize parameters of a web-server.

4.2 Better/Adaptive Designs

Candidate solutions of the traditional black box ADP are configurations ($\Delta \to A$). In our white box formulation, solutions take the form of execution context dependent configurations ($Context \times \Delta \to A$). As a consequence, the white box ADP subsumes the algorithm selection problem. Rather than learning a configuration that is best on average, we learn the best configuration for each context (π^*). Since π^* is independent of the input distribution $\mathcal{D}$, we can re-use white box information in settings where $\mathcal{D}$ might differ/change and can readily use any new experience to improve our policy (e.g. continuous improvement) without the risk of biasing our design.

Remark that a white box approach is not required to solve both algorithm configuration and selection problems simultaneously. Recently, black box solvers have been proposed (e.g. Hydra, ISAC) learning which configuration to use for a given input ($X \times \Delta \to A$). However, the white box approach by nature considers the dynamic variant of the problem: "What configuration to use in a given execution context". As such, we're able to change our design decision dynamically, i.e. different decisions can be made in different execution contexts, even though the input is the same. In addition, black-box methods require a given set of (statically computed) problem features predictive for the performance of any particular configuration. The white box approach on the other hand, can use dynamic features of the resulting execution states. E.g. consider an algorithm taking the seed for a random generator as input (e.g. benchmark 2, Section 6.2). While computing relevant static features from a seed value is clearly troublesome, determining features of the resulting execution states is often trivial (e.g. the current iteration i).

Finally, since the white box ADP considers the superset of adaptive designs, it is potentially more difficult than its black box counter-part, depending on the extent we can exploit white box information to solve it. However, one could restrict the policy space and still exploit white box information to solve the resulting ADP more quickly.

5 A Context-oblivious White Box Optimizer

As explained in Section 3, implementing a (practical) general solver for the context-aware setting is challenging and subject of future research. In this section, we formulate and describe a solver for a "context-oblivious" variant of the ADP to demonstrate that our approach is practical nonetheless. In Section 6, we use this optimizer to illustrate some of the benefits of white box evaluation experimentally.

5.1 The Context-oblivious ADP (COADP)

In the COADP, decisions are made without using execution context information. In what follows, we show that the COADP can also be formulated as an MDP. Here, we further restrict our set of policies to those consistently making the same decision for a design choice during execution. Let $C' = \{c' | c'(z) \in (\kappa(z) \cup \{\theta\}), \forall z \in \Delta\}$ be the set of partial configurations. A design choice z is said to be open in c' if $c'(z) = \theta$. Let $next'(id, c')$ be the ID resulting from simulating TM_{adp} starting from id, choosing $c'(z), \forall z$, until it halts or a design choice open in c' is reached. Formally, $next'(id, c') = next(id) = \alpha z \beta$ if $c'(z) = \theta$ and $next'(id, c') = next'(id', c')$ with $\alpha z \beta \overset{c'(z)}{\vdash} id'$ otherwise.

$MDP_{coadp} = \langle S^{co}, A, T^{co}, R \rangle$, where

S^{co} = $\{(zx, c') \mid \alpha z \beta = next'(q_0 x, c')\} \cup \{s_x | x \in X\} \cup \{\tau\}$.

T^{co} : $S \times A_s \to S$ is the transition function.

$T^{co}(s_x, a_{noop}) = s'$. Let $\alpha z \beta = next(q_0 x)$.
If $z \in F$: $s' = \tau$, otherwise $s' = (zx, c')$ with $c'(dc) = \theta, \forall dc \in \Delta$.

$T^{co}((zx, c'), a) = s'$. Let $\alpha z \beta = next'(q_0 x, c')$ and $\alpha z \beta \overset{a}{\vdash} id'$ and $\alpha' z' \beta' = next'(id', c'')$ with $c''(z) = a$ and $c''(dc) = c'(dc), \forall dc \in \Delta \setminus \{z\}$.
If $z' \in F$: $s' = \tau$, otherwise $s' = (z'x, c'')$.

$T^{co}(\tau, a_{noop}) = \tau$.

Note that the resulting state transition graph is directed acyclic (with exception of the terminal state).

Remark that MDP_{coadp} actually corresponds to a set of independent MDPs, one for each input. As a consequence, this formulation doesn't allow us to generalize observations across inputs directly.

5.2 The Solver

We have implemented our optimizer as a standalone Java Library.[1] It solves the stochastic variant of the context-oblivious ADP (MDP_{coadp}) offline. It doesn't come close to exploiting the full potential laid out in Section 4. Rather, it serves as a Proof of Concept, demonstrating that it is relatively easy to implement and use a basic white box optimizer in practice.

Problem Specification

The programmer statically declares the design choices (Δ) and their respective domains (κ). Users can define any domain they desire by implementing `DesignChoice<T>`. The generic type `T` allows the compiler to guarantee type safety. The library itself already provides some constructors for common domains (e.g. `IntegerInRange`, implementing `DesignChoice<Integer>`). Currently, the optimizer only supports discrete domains. When a decision is required for a design choice (choice point in δ'), a call to `T getDecision(DesignChoice<T>)` will return the decision for that choice. The `feedback(Double)` method is used to reward/penalize (ρ) the agent during execution, indicating the desirability of the current execution path. E.g. if f corresponds to "execution time", the feedback signal (R) corresponds to "the time elapsed between 2 choice points". For some f, assigning credit to parts of the execution might be difficult, in which case potential-based reward-shaping [Ng *et al.*, 1999] can be used (without affecting π^*).

Solution Approach

Our optimizer iteratively evaluates a given algorithm (with open design choices) on a given input (with variable seed). When an open design choice is encountered at runtime, it will query a given agent for a decision. Currently the following agents are supported:

- URS: Selects an alternative uniformly at random.
- PURS: Selects an alternative that was not yet selected uniformly at random, if any. Otherwise, an alternative a is selected proportional to the expected number of execution branches $N(s, a) = \sum_{s'} P(s, a, s') N(s')$ with $N(s) = \sum_a N(s, a)$ and $N(f) = 1$.
- GR: Pure greedy, selects the alternative estimated to maximize the expected future reward.

When testing, we wish to use an agent that follows an optimal policy (i.e. GR). However, during the training phase the agent should explore alternative design decisions (e.g. (P)URS), i.e. guide the search.

During these evaluations, the following information about state transitions is recorded:

- $n(s, a, s')$: The number of times taking decision a in state s led to state s'.
- $r(s, a, s')$: The average reward received after taking decision a in state s, before ending up in the next state s'.

To reduce space requirements, these were only stored for the s, a, s' encountered and assumed to be 0 otherwise. This information is used to approximate P^{co} and R of MDP_{coadp}:

$$\tilde{P}^{co}(s, a, s') = \frac{n(s, a, s')}{\sum_{s''} n(s, a, s'')} \qquad \tilde{R}(s, a, s') = r(s, a, s')$$

Finally, the optimizer assumes the best design, at any time, to be the policy $\tilde{\pi}^*$ solving $\widetilde{MDP}_{coadp} = \langle S^{co}, A, \tilde{P}^{co}, \tilde{R} \rangle$. Here $\widetilde{MDP}_{coadp}$ is solved using a bottom-up dynamic programming approach and ties are broken randomly. The idea is that multiple evaluations will allow us to accurately approximate the actual MDP and therefore its optimal policy.

Note that $\widetilde{MDP}_{coadp}$ is a consistent estimator for MDP_{coadp} as long as each policy π has a positive chance of being explored. Our optimizer is therefore Probabilistically Approximately Correct [Valiant, 1984] as long as the agent used has this guarantee (e.g. (P)URS).

[1] https://github.com/Steven-Adriaensen/White-box-ADP

6 Experimental Illustrations

In this section, we illustrate some of the benefits of white box evaluation. To this purpose, we compare the performance of the implementation described in Section 5 to that of a similar black box implementation on two micro-benchmarks. These implementations differ in that the white box optimizer (WB) maintains transition data (n,r) and returns $\tilde{\pi}^*$, while the black box optimizer (BB) maintains a $c \rightarrow \bar{f}(c)$ mapping and returns $\tilde{c}^* = \arg\max_{c'} \bar{f}(c')$.

Both optimizers perform Uniform Random Sampling. URS was chosen to keep things simple and illustrate the contribution of using white box information *by itself*. On some problems other features of state-of-the-art configurators (e.g. local search) might render white box benefits to some extent redundant. In general, white box evaluation is orthogonal to these more advanced sampling schemes and can be used to inform them (c.f. PURS for URS). Our experiments are by no means intended as a comparison of/to the state-of-the-art (WB nor BB are). The benchmarks have been selected to illustrate the benefits of our white box approach. We make no claims about their representativeness for real-world problems.

Figures 2 and 4 show the performance of the algorithm returned by each optimizer, after x algorithm evaluations,[2] averaged over 100 independent meta-optimization runs. Here we distinguish between the actual performance (full line) and the performance as estimated by the optimizer (dashed line).

6.1 Benchmark 1: Speed up Meta-optimization

Design Choices:
`<Integer> dc[i] = {0,1} for 1 ≤ i ≤ 20`

```
i = 1;
while i ≤ 20 and getDecision(dc[i]) = 1 do
   feedback(random_gaussian(1,2));
   i = i + 1;
end while
```

Figure 1: Code for Benchmark 1

Figure 1 shows the code for benchmark 1. In this benchmark, we are to make up to 20 design decisions between 0 and 1, in a fixed order. As long as we choose 1, we receive a reward drawn from $\mathcal{N}(1,2)$, otherwise execution terminates. While there are 2^{20} different configurations, there are only 21 different execution paths, and $\pi^*(s) = 1, \forall s$.

Figure 2 compares our WB optimizer using the URS and PURS agents to the BB optimizer (BB-URS). We observe that each method overestimates the performance of its incumbent design. This schism is known as overconfidence (*oc*) [Birattari and Dorigo, 2004] and is caused by the fact that the maximum sample average is a biased estimator. Due to *oc*, an optimizer cannot distinguish bad from good designs, causing unreliable performance. BB-URS clearly exhibits most *oc* and performs the worst. It is only after about 1M evaluations that *oc* drops as configurations are re-evaluated. The WB optimizers have a much lower estimation error, as they are able

[2] An evaluation is a single run with a variable seed as only input.

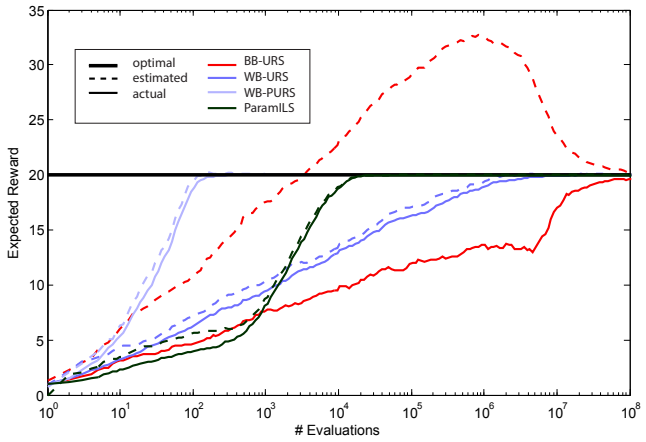

Figure 2: Results for Benchmark 1

to generalize across configurations. By reusing information obtained during the evaluation of similar policies, they improve their estimate of the performance of the best policy so far. Both URS methods take a long time, given there are only 21 execution paths. This is because, even though policies are sampled uniformly at random, execution paths are not. E.g. there is only a single configuration corresponding to the best execution path, but 2^{19} to the worst. On average, WB-URS finds the optimal solution after $\pm$ 1M evaluations, which is the best we could hope for. However, using white box evaluation, we obtain information about how design decisions influence future choice points, i.e. alternative execution paths, which we can use to inform the sampling. WB-PURS attempts to draw samples uniformly w.r.t. execution paths, and as there are only 21 different ones, it quickly finds the best ($\pm$100 evaluations). To put this in perspective, ParamILS[3] required $\pm$ 10K evaluations on this benchmark.

6.2 Benchmark 2: Better/Adaptive Designs

Design Choices:
`<Integer> dc[i] = {1,2,3,4,5} for 1 ≤ i ≤ 5`

```
order = shuffle([1,2,3,4,5]);
for 1 ≤ i ≤ 5 do
   if getDecision(dc[order[i]])= i then
      feedback(1);
   end if
end for
```

Figure 3: Code for Benchmark 2

Figure 3 shows the code for benchmark 2. In this benchmark we are to make 5 design decisions between 1, 2, 3, 4 and 5, in a random order. If we choose alternative i the i^{th} design choice (iteration), we receive a reward of 1. Here, the best decision depends on the number of design decisions made before, which is only known at runtime.

[3] The *focused* variant, using the default parameter settings.

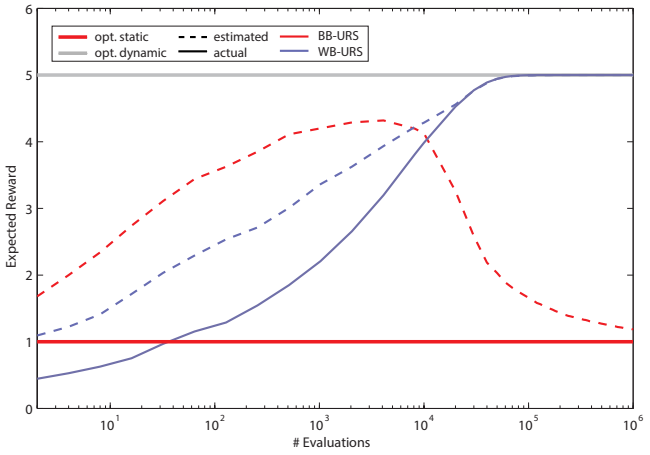

Figure 4: Results for Benchmark 2

Figure 4 shows that WB-URS finds the optimal dynamic policy, which accumulates a reward of 5. This might surprise, since our optimizer is context oblivious. However, S^{co} does include minimal runtime information (partial configuration $c' \in C'$), which is sufficient to act optimally in this benchmark. BB-URS only considers static configurations, which are all as good, accumulating an expected reward of 1. Note that BB-URS also suffers from *oc* on this benchmark.

7 Related Research

This section lists prior-art on algorithm design, that can be viewed as following a white box approach. Rather than being complete, we attempt to characterize their nature and clarify how our work differs from theirs.

We consider designing algorithms in general (TM_{adp}). Prior-art considers specific subclasses of algorithms (e.g. Divide & Conquer-algorithms [Lagoudakis and Littman, 2000], Evolutionary Algorithms [Eiben *et al.*, 1999; Karafotias *et al.*, 2014], Meta-heuristics [Burke *et al.*, 2003], Constraint Satisfaction Programs [Epstein *et al.*, 2002] etc). We reformulate this problem as an MDP. While some (e.g. [Lagoudakis and Littman, 2000; Karafotias *et al.*, 2014]) informally suggest this could be done, they eventually formulate and solve a problem that is not Markovian. The Markov-property is what enables us to generalize across runs, inputs and configurations in a theoretically sound manner. Also, note that the MDP formulation in [Lagoudakis and Littman, 2000] is far more black-box than ours (states represent independent sub-problems, actions correspond to algorithms).

Finally, there exists a vast body of research attempting to make/adapt design decisions at runtime. Well-known examples are reactive-search [Battiti *et al.*, 2008] and selection hyper-heuristics [Burke *et al.*, 2003]. Online adaptivity is indeed a key advantage of the white box approach, enabling better designs (see Section 4.2). However, to the best of our knowledge, prior-art does this outside of a proper theoretical framework, relying mostly on heuristics. Learning, if used at all, is done online, without generalization across runs and introduces a trade-off between exploration and exploitation. Benefits as such are often minimal and exploitation of learned knowledge requires very long/repetitive runs (e.g. an RL approach performs barely better than random in [Karafotias *et al.*, 2014]). Furthermore, the RL problem these methods (often implicitly) solve, has a non-stationary environment, thus requiring us to learn faster than the environment changes. A proper theoretical framework is exactly what we propose in this paper. Note that in our formulation, online-learning is no prerequisite for online adaptivity, i.e. we can learn "what works when" offline and utilize this knowledge online.

8 Conclusion

In summary, this paper focuses on formulating the Algorithm Design Problem (ADP) in a general and formal manner (TM_{adp}), describing/illustrating its potential benefits, without bias towards a particular solution approach, something prior-art, without exception, fails to do. The ADP as formulated in this work subsumes the algorithm selection, parametrization and dynamic adaptation problems, and as such might be foundational to enabling a more unified approach to automated algorithm design in the future. The MDP_{adp} formulation is of particular interest, as it allows us to leverage the gamut of MDP solution approaches (e.g. RL) that have been proposed over the past six decades to do so.

While exploiting the full potential of the white box approach is a challenging endeavor and was beyond the scope of this paper, we believe that the envisioned benefits should make this a research track worth exploring. Furthermore, to convince the reader our approach is practical nonetheless, we have presented a concrete white box optimizer and used it to illustrate how white box information can be utilized to speed up the optimization process as well as improve the quality of the resulting design, on two illustrative micro-benchmarks. Subject of future research is a white box optimizer, outperforming state-of-the-art algorithm configurators on real-world ADPs. Worst-case, if each configuration uniquely affects the execution on every input, we can do no better. Our objective is therefore to never do worse. Also subject of future research is a practical approach to profit from the (partial) observability of context during the design process. Finally, we wish to explore the potential of using this framework to analyze and compare heuristics (human-defined policies) commonly used to make design choices at runtime and how these can be used to inform the optimization process.

In the future, automated algorithm design will play an increasingly prominent role. Recently, multiple sources [Ansel *et al.*, 2009; Hoos, 2012; Adriaensen *et al.*, 2014] have independently advocated a PbO-like methodology to design algorithms. Simultaneously, many fields have in recent years suffered from an uncontrolled introduction of algorithmic variation. We expect that maturation of these fields will lead to a more controlled, modular, component-driven approach (e.g. Metaheuristic Optimization [Swan *et al.*, 2015]). The question of which components to combine to solve a given problem will surely be one computers must help answering.

Acknowledgments

Currently, Steven Adriaensen is funded by a Ph.D grant of the Research Foundation Flanders (FWO).

References

[Adriaensen *et al.*, 2014] Steven Adriaensen, Tim Brys, and Ann Nowé. Designing reusable metaheuristic methods: A semi-automated approach. In *Evolutionary Computation (CEC), 2014 IEEE Congress on*, pages 2969–2976. IEEE, 2014.

[Ansel *et al.*, 2009] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. Petabricks: A language and compiler for algorithmic choice. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Dublin, Ireland, June 2009.

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

[Battiti *et al.*, 2008] Roberto Battiti, Mauro Brunato, and Franco Mascia. *Reactive search and intelligent optimization*, volume 45. Springer Science & Business Media, 2008.

[Bellman, 1957] Richard Bellman. A markovian decision process. Technical report, DTIC Document, 1957.

[Birattari and Dorigo, 2004] Mauro Birattari and Marco Dorigo. *The problem of tuning metaheuristics as seen from a machine learning perspective*. PhD thesis, 2004.

[Burke *et al.*, 2003] Edmund Burke, Graham Kendall, Jim Newall, Emma Hart, Peter Ross, and Sonia Schulenburg. Hyper-heuristics: An emerging direction in modern search technology. *International series in operations research and management science*, pages 457–474, 2003.

[Eiben *et al.*, 1999] Agoston Endre Eiben, Robert Hinterding, and Zbigniew Michalewicz. Parameter control in evolutionary algorithms. *Evolutionary Computation, IEEE Transactions on*, 3(2):124–141, 1999.

[Epstein *et al.*, 2002] Susan L Epstein, Eugene C Freuder, Richard Wallace, Anton Morozov, and Bruce Samuels. The adaptive constraint engine. In *Principles and Practice of Constraint Programming-CP 2002*, pages 525–540. Springer, 2002.

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

[Hopcroft, 1979] John E Hopcroft. *Introduction to automata theory, languages, and computation*. Pearson Education India, 1979.

[Howard, 1960] Ronald A. Howard. *Dynamic Programming and Markov Processes*. The M.I.T. Press, 1960.

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

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

[Kadioglu *et al.*, 2010] Serdar Kadioglu, Yuri Malitsky, Meinolf Sellmann, and Kevin Tierney. ISAC-Instance-Specific algorithm configuration. In *ECAI*, volume 215, pages 751–756, 2010.

[Kaelbling *et al.*, 1998] Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra. Planning and acting in partially observable stochastic domains. *Artificial intelligence*, 101(1):99–134, 1998.

[Karafotias *et al.*, 2014] Giorgos Karafotias, Agoston Endre Eiben, and Mark Hoogendoorn. Generic parameter control with reinforcement learning. In *Proceedings of the 2014 conference on Genetic and evolutionary computation*, pages 1319–1326. ACM, 2014.

[Koza, 1992] John R Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

[Lagoudakis and Littman, 2000] Michail G Lagoudakis and Michael L Littman. Algorithm selection using reinforcement learning. In *ICML*, pages 511–518. Citeseer, 2000.

[López-Ibánez *et al.*, 2011] Manuel López-Ibánez, Jérémie Dubois-Lacoste, Thomas Stützle, and Mauro Birattari. The irace package, iterated race for automatic algorithm configuration. Technical report, Universit Libre de Bruxelles, 2011.

[Ng *et al.*, 1999] Andrew Y Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *ICML*, volume 99, pages 278–287, 1999.

[Rice, 1976] John R Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976.

[Sutton and Barto, 1998] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[Swan *et al.*, 2015] Jerry Swan, Steven Adriaensen, Mohamed Bishr, Edmund K Burke, John A Clark, Patrick De Causmaecker, Juanjo Durillo, Kevin Hammond, Emma Hart, Colin G Johnson, et al. A research agenda for metaheuristic standardization. In *MIC 2015: The 11th Metaheuristics International Conference*, 2015.

[Turing, 1936] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

[Valiant, 1984] Leslie G Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.

[Wiering and van Otterlo, 2012] Marco Wiering and Martijn van Otterlo. *Reinforcement Learning: State-of-the-art*, volume 12. Springer Science & Business Media, 2012.

[Wolpert and Macready, 1997] David H Wolpert and William G Macready. No free lunch theorems for optimization. *Evolutionary Computation, IEEE Transactions on*, 1(1):67–82, 1997.

[Xu *et al.*, 2010] Lin Xu, Holger Hoos, and Kevin Leyton-Brown. Hydra: Automatically configuring algorithms for portfolio-based selection. In *AAAI*, volume 10, pages 210–216, 2010.