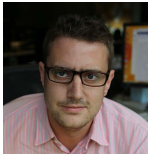
Algorithm Synthesis with Theoretical Guarantees

Kevin Leyton-Brown

University of British Columbia
Distinguished University Scholar
Canada CIFAR AI Chair, Amii

THE UNIVERSITY
OF BRITISH COLUMBIA

This talk surveys work with many amazing coauthors!



Devon Graham

- **Eros Rojas**

- András György
- Bobby Kleinberg
- Wei-I Lin

- **Tim Roughgarden**

- Brendan Lucier
- Csaba Szepesvári
- Gellert Weisz

Talk Overview

Algorithm Configuration

Algorithm Configuration with Theoretical Guarantees

Reconsidering the Objective Function

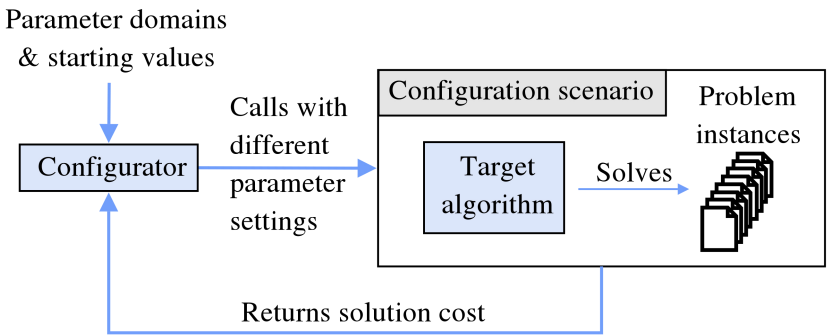
Utilitarian Algorithm Configuration

Empirical Evaluation

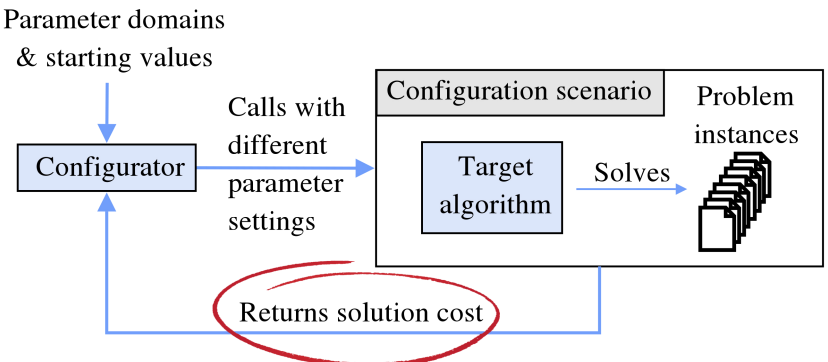
Our Focus Today

- You face a **computationally hard problem**
- Lots of **alternative solution strategies** are available
  - many or even all solve the problem correctly given enough time
  - they vary dramatically in their runtimes
  - which one performs best depends on the inputs given (and on what you mean by “best”)
- There are many **examples** of problems like this:
  - mixed-integer programming
  - SAT solving
  - chain-of-thought reasoning to solve Math Olympiad problems
- **Algorithm configuration**: choose a solution strategy based on data
- **Theoretical guarantees**:
  - how well does the result perform on the true distribution that generated the data?
  - how efficiently does the algorithm configuration procedure search the space of alternatives?

Algorithm Configuration Visualized



Algorithm Configuration Visualized



Heuristic Algorithm Configuration Procedures

A wide variety of **practical algorithm configuration procedures** have been proposed

- **F-race** [Birattari, Stützle, Paquete & Varrentrapp, 2002];
  irace [López-Ibáñez, Dubois- Lacoste, Stützle & Birattari, 2011]: racing
- **ParamILS** [Hutter, Hoos, & Stützle, 2007; Hutter, Hoos, Leyton-Brown & Stützle 2009]: local search
- **GGA** [Ansótegui, Sellmann & Tierney, 2009; 2015]: genetic algorithms
- **SMAC** [Hutter, Hoos & Leyton-Brown, 2011]: Bayesian optimization

Sequential Model-based Algorithm Configuration (SMAC)

[Hutter, Hoos & L-B, 2011]

Learn a model from performance data so far;
Use model to select promising new configurations;

Sequential Model-based Algorithm Configuration (SMAC)

[Hutter, Hoos & L-B, 2011]

Learn a model from performance data so far;
Use model to select promising new configurations;
Perform new algorithm runs to compare new configurations to best configuration so far

Sequential Model-based Algorithm Configuration (SMAC)

[Hutter, Hoos & L-B, 2011]

Initialize by executing some runs and collecting their performance data;

repeat

- Learn a model from performance data so far;
- Use model to select promising new configurations;
- Perform new algorithm runs to compare new configurations to best configuration so far

until *time budget exhausted*;

Talk Overview

Algorithm Configuration

Algorithm Configuration with Theoretical Guarantees

Reconsidering the Objective Function

Utilitarian Algorithm Configuration

Empirical Evaluation

Are Heuristics Enough?

- SMAC maintains an **incumbent** configuration that looks best so far
- Its core logic is driven by heuristic stand-ins for statistical tests
  - Aggressively discard **challenger** configurations if they don't quickly outperform
  - Never stop running the incumbent, even when we understand its performance very well
- Often **works well in practice**

What if we approach this problem with a proper theoretical grounding?

Algorithm Configuration with Optimality Guarantees

Can we make meaningful guarantees about the quality of the returned configuration, not just on the samples used to evaluate it but on the true distribution?

A Guarantee

The average runtime of the returned configuration is within ϵ of the best configuration.

Algorithm Configuration with Optimality Guarantees

Can we make meaningful guarantees about the quality of the returned configuration, not just on the samples used to evaluate it but on the true distribution?

A Guarantee

The average runtime of the returned configuration is within ϵ of the best configuration.

Runtimes can be unbounded. What if **average runtime is infinite?**

Algorithm Configuration with Optimality Guarantees

Can we make meaningful guarantees about the quality of the returned configuration, not just on the samples used to evaluate it but on the true distribution?

A Guarantee

The **capped** average runtime of the returned configuration is within ϵ of the best configuration, **and this capping only affected a β-fraction of instances.**

Algorithm Configuration with Optimality Guarantees

Can we make meaningful guarantees about the quality of the returned configuration, not just on the samples used to evaluate it but on the true distribution?

A Guarantee

The capped average runtime of the returned configuration is within ϵ of the best configuration, and this capping only affected a β-fraction of instances.

What if the configuration space is **huge or unbounded?**

Algorithm Configuration with Optimality Guarantees

Can we make meaningful guarantees about the quality of the returned configuration, not just on the samples used to evaluate it but on the true distribution?

$(\epsilon, \beta, \gamma)$-Optimality

The capped average runtime of the returned configuration is within ϵ of the best configuration, this capping only affected a β-fraction of instances, **and we sampled enough configurations that the probability of sampling anything better is no greater than γ.**

Algorithm Configuration with Optimality Guarantees

Can we make meaningful guarantees about the quality of the returned configuration, not just on the samples used to evaluate it but on the true distribution?

$(\epsilon, \beta, \gamma)$-Optimality

The capped average runtime of the returned configuration is within ϵ of the best configuration, this capping only affected a β-fraction of instances, and we sampled enough configurations that the probability of sampling anything better is no greater than γ.

Theorem [Kleinberg, Leyton-Brown & Lucier, 2017]

Any procedure that proves $(\epsilon, \beta, \gamma)$-optimality requires worst case time T at least $T \geq \frac{\text{LOGTERM}}{\epsilon^2 \beta} OPT_\gamma$, where OPT_γ is the runtime of the best configuration remaining after excluding the top γ-fraction of configurations.

Algorithm Configuration with Optimality Guarantees

Some existing configuration procedures offer **optimality guarantees**:

Existing procedures

- Structured Procrastination [Kleinberg, Leyton-Brown & Lucier, IJCAI 2017]
- LeapsAndBounds [Weisz, György & Szepesvári, ICML 2018]
- CapsAndRuns [Weisz, György & Szepesvári, ICML 2019]
- Structured Procrastination with Confidence [Kleinberg, Leyton-Brown, Lucier & Graham, NeurIPS 2019]
- ImpatientCapsAndRuns [Weisz, György, Lin, Graham, Leyton-Brown & Szepesvári, NeurIPS 2020]
- AC-Band [Brandt, Schede, Haddenhorst, Bengs, Hüllermeier & Tierney, AAAI 2023]

All guarantee **nearly optimal** performance: within a log factor of the lower bound.

All spend lots of time ensuring that the best-performing configurations don't very occasionally have very long runs.

Talk Overview

Algorithm Configuration

Algorithm Configuration with Theoretical Guarantees

Reconsidering the Objective Function

Utilitarian Algorithm Configuration

Empirical Evaluation

Which algorithm do you prefer?

Consider a family of algorithms that always return the right answer but vary in their runtimes, like a set of SAT solvers. Much existing work assumes that the best algorithm is the one with the **best average runtime**. But is it really?

Which algorithm do you prefer?

Consider a family of algorithms that always return the right answer but vary in their runtimes, like a set of SAT solvers. Much existing work assumes that the best algorithm is the one with the **best average runtime**. But is it really?

Two Algorithms

Algorithm A_1

- Solves 99 instances in 1 second.
- Runs the 100$^{\text{th}}$ instance for 10 days but fails to solve it.

Algorithm A_2

- Runs all 100 instances for 10 days each but fails to solve any.

Problem 1

Average runtimes are completely unconstrained by this information. Does that mean that we don't know enough to have any preference between the algorithms?

Which algorithm do you prefer?

Consider a family of algorithms that always return the right answer but vary in their runtimes, like a set of SAT solvers. Much existing work assumes that the best algorithm is the one with the **best average runtime**. But is it really?

Two Algorithms

Algorithm A_1

- Solves 99 instances in 1 second.
- Runs the 100$^{\text{th}}$ instance for 10 days but fails to solve it.

Algorithm A_2

- Runs all 100 instances for 10 days each but fails to solve any.

Problem 2

Imagine that both algorithms contain a bug, and the long runs never terminate. Then both averages are infinite. In this case, are we indifferent between A_1 and A_2?

Handling Capped Runs

- We often appear able to **prefer one algorithm over another** even when some runs are stopped early ("capped")
- But how exactly should we account for such runs?
  - consider only the **fraction of problems solved**
  - consider capped runs to have **completed at the captime** (PAR1)
  - consider capped runs to have **completed at $k > 1$ times the captime** (PARk)
- Relative rankings between algorithms depend critically on choice of captime; k
- We will describe our preferences using a **utility function**

Preferences and Utility

A first-principles investigation of our preferences implies that they must be described by some utility function.

Axioms

- **Von Neumann–Morgenstern** axioms [Von Neumann & Morgenstern, 1947]
- **Eagerness** (“short runs are at least as good as long runs”)
- **Relevance** (“solves are strictly preferable to timeouts”)

Decision-theory basics

Runtime-specific from our paper

Theorem [Graham, Leyton-Brown & Roughgarden, ICML 2023]

If our preferences satisfy the axioms, then there exists a utility function u for which

$$i \text{ is preferred to } j \iff \text{expected utility of } i > \text{expected utility of } j.$$

A Concrete Example - Known Captime

Example (linear value for money; pay for compute)

We are risk neutral and have a linear value for money.

We face no explicit runtime cap but we have to buy our compute on Amazon EC2.

- Solving the problem is worth v
- Each hour of compute costs α
- We can avoid negative payoffs by using captime $\kappa^* = v/\alpha$

$\Longrightarrow$

$$u(t) = \max(1 - \tfrac{\alpha t}{v}, 0)$$

"The best algorithm costs least on average."

"Which algorithm is better?"

Algorithm A_1

- Solves 99 instances in 1 second.
- Runs the 100$^{\text{th}}$ instance for 10 days but fails to solve it.

Algorithm A_2

- Runs all 100 instances for 10 days each but fails to solve any.

A_1 preferred to A_2 if $v/\alpha < 10 \cdot 24$ hours.

A Concrete Example - Known Captime Distribution

Example (step-function utility, unknown captime)

Our client will demand an answer at some point in the future, described by K

- Step-function utility *(i.e., $u(t, \kappa) = 1$ for all $t < \kappa$)*
- **Unknown captime ($\kappa \sim K$)**

$\Longrightarrow$

$$u(t) = \Pr_\kappa(t \leq \kappa)$$

"The best algorithm is the one most likely to finish before the captime, **in expectation** over captimes."

"Which algorithm is better?"

Algorithm A_1

- Solves 99 instances in 1 second.
- Runs the 100th instance for 10 days but fails to solve it.

Algorithm A_2

- Runs all 100 instances for 10 days each but fails to solve any.

A_1 preferred to A_2 if κ is always at least 1 second and there is even just a 1% chance that κ will be less than 10 days.

Underspecified Distribution: The Method of Maximum Entropy

What if we know only **constraints on the distribution of κ**?

Principle of Maximum Entropy

If we do not know which distribution to use among some set of alternatives, we should **use the one having greatest entropy**, since it is the least informative and thus incorporates no extraneous assumptions.

Maximizing entropy "spreads out" the distribution's probability mass as much as the conditions allow

Example (Bounded interval)

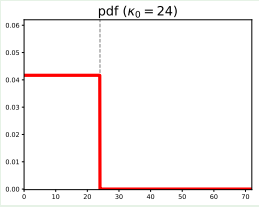
If all we know is that the distribution has support on some bounded interval, the uniform distribution has maximum entropy because it is "flattest."

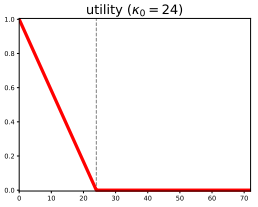
A Concrete Example - Underspecified Distribution

Example (Bounded time interval)

Our client will need a solution to their SAT problem sometime in the next 24 hours.

The maximum entropy distribution is uniform:





The utility function is the survival distribution for this pdf ($P(t \geq x)$)

$$u(t) = \begin{cases} 1 - \frac{t}{24h} & \text{if } t < 24h \\ 0 & \text{otherwise} \end{cases}$$

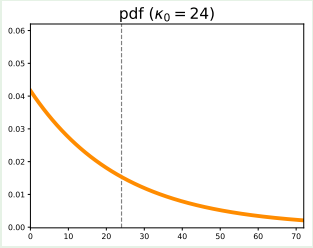
(linear utility function)

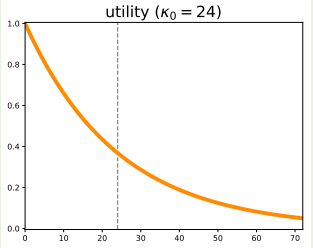
A Concrete Example - Underspecified Distribution

Example (Known mean)

We do not know how long the client will give us to solve their SAT problem but the expected value of the captime distribution is 24 hours.

***The maximum entropy distribution is exponential*:**





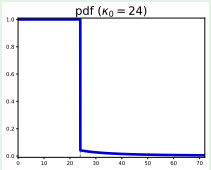
$$u(t) = e^{-\frac{t}{24h}}$$

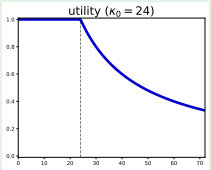
(exponential utility function)

A Concrete Example - Underspecified Distribution

Example (Known geometric mean)

We know we'll always have at least one day, and we assume that the geometric mean of the captime will be e times that minimum ($\mathbb{E}[\ln(t/24h)] = 1$). The maximum entropy distribution is Pareto:





$$u(t) = \begin{cases} 1 & \text{if } t \leq 24h \\ \frac{24h}{t} & \text{otherwise} \end{cases}$$

(Pareto utility function)

Interpretation: We are indifferent between runs that take less than 24h; beyond this point, every doubling of runtime leads to a halving of our utility.

Explicit Utility Function

We can also express a utility function directly. This can be particularly useful if we want to **trade off solution quality and runtime**.

Example (Pareto for runtime; linear for solution quality)

We get utility of 1 for runs that have solution quality 1 and that finish in 1h or less. Holding solution quality fixed, every doubling of runtime leads to a halving of our utility. Holding runtime fixed, we lose 0.1 utility for every solution quality drop of 0.01, reaching utility 0 at solution quality 0.9 or below.

$$u(t,q) = \begin{cases} 10 \cdot \max(q - 0.9, 0) & \text{if } t \leq 1h \\ \frac{1h}{t} \cdot 10 \cdot \max(q - 0.9, 0) & \text{otherwise} \end{cases}$$

Talk Overview

Algorithm Configuration

Algorithm Configuration with Theoretical Guarantees

Reconsidering the Objective Function

Utilitarian Algorithm Configuration

Empirical Evaluation

Utilitarian Algorithm Configuration

- Optimizing utility is **fundamentally different** than optimizing average runtime
  - **Maximization objective where utility decreases with time**: most utility comes from short runs
  - Compare to **minimization objective where average runtime grows unboundedly**: an incumbent that looks good could actually be bad if rare, bad runs dominate the average
- We need **new algorithms** to target utilitarian configuration
- Good news: this problem is easier, and the **new algorithms work better**!
- Easier to achieve **theoretical guarantees** too
  - nontrivial upper and lower bounds on utility throughout
  - no need for a parameter β to bound the fraction of capped instances
  - just use our utility bounds directly!

Optimistic Utilitarian Procrastination

OUP: UCB version [Graham, LB & Roughgarden, NeurIPS 2023; Graham & LB, ICLR 2025]

initialize $\kappa_c = 1$ for each configuration $c \in C$ // initial captime; choice of units up to the user

initialize $U_c = 1$ and $L_c = 0$ for each configuration $c \in C$ // initial bounds on expected utility from c

Optimistic Utilitarian Procrastination

OUP: UCB version [Graham, LB & Roughgarden, NeurIPS 2023; Graham & LB, ICLR 2025]

initialize $\kappa_c = 1$ for each configuration $c \in C$ // initial captime; choice of units up to the user
initialize $U_c = 1$ and $L_c = 0$ for each configuration $c \in C$ // initial bounds on expected utility from c
repeat

until interrupted

Optimistic Utilitarian Procrastination

OUP: UCB version [Graham, LB & Roughgarden, NeurIPS 2023; Graham & LB, ICLR 2025]

initialize $\kappa_c = 1$ for each configuration $c \in C$ // initial captime; choice of units up to the user
initialize $U_c = 1$ and $L_c = 0$ for each configuration $c \in C$ // initial bounds on expected utility from c
repeat
    select configuration $c^* \in \arg\max_c U_c$ // configuration with largest upper bound
    run c^* with captime κ_{c^*} on a new instance

until interrupted

Optimistic Utilitarian Procrastination

OUP: UCB version [Graham, LB & Roughgarden, NeurIPS 2023; Graham & LB, ICLR 2025]

initialize $\kappa_c = 1$ for each configuration $c \in C$ // initial captime; choice of units up to the user
initialize $U_c = 1$ and $L_c = 0$ for each configuration $c \in C$ // initial bounds on expected utility from c
repeat
    select configuration $c^* \in \arg\max_c U_c$ // configuration with largest upper bound
    run c^* with captime κ_{c^*} on a new instance
    update U_{c^*}, L_{c^*}

until interrupted

Optimistic Utilitarian Procrastination

OUP: UCB version [Graham, LB & Roughgarden, NeurIPS 2023; Graham & LB, ICLR 2025]

initialize $\kappa_c = 1$ for each configuration $c \in C$ // initial captime; choice of units up to the user
initialize $U_c = 1$ and $L_c = 0$ for each configuration $c \in C$ // initial bounds on expected utility from c
repeat
    select configuration $c^* \in \arg\max_c U_c$ // configuration with largest upper bound
    run c^* with captime κ_{c^*} on a new instance
    update U_{c^*}, L_{c^*}
    for each $c \in C$ **if** doubling condition is met for c, **double** κ_c and rerun all capped runs of c
until interrupted

Optimistic Utilitarian Procrastination

OUP: UCB version [Graham, LB & Roughgarden, NeurIPS 2023; Graham & LB, ICLR 2025]

initialize $\kappa_c = 1$ for each configuration $c \in C$ // initial captime; choice of units up to the user
initialize $U_c = 1$ and $L_c = 0$ for each configuration $c \in C$ // initial bounds on expected utility from c
repeat
    select configuration $c^* \in \arg\max_c U_c$ // configuration with largest upper bound
    run c^* with captime κ_{c^*} on a new instance
    update U_{c^*}, L_{c^*}
    for each $c \in C$ **if** doubling condition is met for c, **double** κ_c and rerun all capped runs of c
until interrupted
return $\arg\max_c L_c$ // configuration with largest lower bound

Optimistic Utilitarian Procrastination

OUP: UCB version [Graham, LB & Roughgarden, NeurIPS 2023; Graham & LB, ICLR 2025]

initialize $\kappa_c = 1$ for each configuration $c \in C$ // initial captime; choice of units up to the user
initialize $U_c = 1$ and $L_c = 0$ for each configuration $c \in C$ // initial bounds on expected utility from c
repeat
  select configuration $c^* \in \arg\max_c U_c$ // configuration with largest upper bound
  run c^* with captime κ_{c^*} on a new instance
  update U_{c^*}, L_{c^*}
  for each $c \in C$ **if** doubling condition is met for c, **double** κ_c and rerun all capped runs of c
until interrupted
return $\arg\max_c L_c$ // configuration with largest lower bound

- **Fact:** Gap between upper and lower confidence bounds decomposes into two terms: $g_c + g_u$ depending on capped and uncapped runs.
- **Doubling Condition:** $g_c > g_u$.

Optimistic Utilitarian Procrastination

OUP: UCB version [Graham, LB & Roughgarden, NeurIPS 2023; Graham & LB, ICLR 2025]

initialize $\kappa_c = 1$ for each configuration $c \in C$ // initial captime; choice of units up to the user
initialize $U_c = 1$ and $L_c = 0$ for each configuration $c \in C$ // initial bounds on expected utility from c
repeat
    select configuration $c^* \in \arg\max_c U_c$ // configuration with largest upper bound
    run c^* with captime κ_{c^*} on a new instance
    update U_{c^*}, L_{c^*}
    for each $c \in C$ **if** doubling condition is met for c, **double** κ_c and rerun all capped runs of c
until interrupted
return $\arg\max_c L_c$ // configuration with largest lower bound

- **Fact:** Gap between upper and lower confidence bounds decomposes into two terms: $g_c + g_u$ depending on capped and uncapped runs.
- **Doubling Condition:** $g_c > g_u$.
- **Tighter bounds:** Instead of using Hoeffding's inequality, as most UCB implementations do, appeal to Chernoff's Lemma directly and solve for bounds numerically [cf. Lattimore & Szepesvári 2020]

OUP-UCB: Analysis

- OUP-UCB is an instance of the UCB algorithm [Lai, 1987], implying that provably:
  - With high probability, the returned configuration has **expected utility within an additive ϵ of optimal**, $\epsilon = \max_c U_c - \max_{c'} L_{c'}$
  - runs until terminated, **improving ϵ as it runs**
  - takes **no more samples than necessary** in the worst case
  - performs **better on easier inputs**
- OUP's running time is efficient:
  - no more than roughly twice the runtime we would have had **if we started with the final κs**
    ("roughly": there's an extra $O(\log \log \kappa)$ term that arises from the union bound over multiple captimes)

OUP: Better Performance via LUBC

The LUCB "Lower Upper Confidence Bound" algorithm [Kalyanakrishnan, Tewari, Auer & Stone, 2012] offers better empirical performance for "best arm identification" than UCB

OUP: UCB Version [Graham, LB & Roughgarden, NeurIPS 2023; Graham & LB, ICLR 2025]

initialize $\kappa_c = 1$ for each configuration $c \in C$ // initial captime; choice of units up to the user
initialize $U_c = 1$ and $L_c = 0$ for each configuration $c \in C$ // initial bounds on expected utility from c
repeat
    select configuration $c^* \in \arg\max_c U_c$ // configuration with largest upper bound
    run c^* with captime κ_{c^*} on a new instance
    update U_{c^*}, L_{c^*}
    for each $c \in C$ **if** doubling condition is met for c, **double** κ_c and rerun all capped runs
until interrupted
return $\arg\max_c L_c$ // configuration with largest lower bound

OUP: Better Performance via LUBC

The LUCB "Lower Upper Confidence Bound" algorithm [Kalyanakrishnan, Tewari, Auer & Stone, 2012] offers better empirical performance for "best arm identification" than UCB

OUP: LUCB Version [Graham, LB & Roughgarden, NeurIPS 2023; Graham & LB, ICLR 2025; Graham, Rojas & LB, AAAI 2026]

initialize $\kappa_c = 1$ for each configuration $c \in C$ // initial captime; choice of units up to the user
initialize $U_c = 1$ and $L_c = 0$ for each configuration $c \in C$ // initial bounds on expected utility from c
repeat
    select configuration $c_1 \in C$ having max average utility across all runs performed so far
    select configuration $c_2 \in \arg\max_{c \in C \setminus \{c_1\}} U_c$ // configuration other than c_1 with largest upper bound
    run c_1, c_2 with captimes κ_{c_1}, κ_{c_2}, each on one fresh instance
    update $U_{c_1}, U_{c_2}, L_{c_1}, L_{c_2}$
    for each $c \in C$ **if** doubling condition is met for c, **double** κ_c and rerun all capped runs
until interrupted
return $\arg\max_c L_c$ // configuration with largest lower bound

Continuous Optimistic Utilitarian Procrastination

OUP (Optimistic Utilitarian Procrastination): LUCB Version [Graham, Rojas & LB, AAAI 2026]

initialize $\kappa_c = 1$ for each configuration $c \in C$ // initial captime; choice of units up to the user
initialize $U_c = 1$ and $L_c = 0$ for each configuration $c \in C$ // initial bounds on expected utility from c
repeat
  select configuration $c_1 \in C$ having max average utility across all runs performed so far
  select configuration $c_2 \in \arg\max_{c \in C \setminus \{c_1\}} U_c$ // configuration other than c_1 with largest upper bound
  run c_1, c_2 with captimes κ_{c_1}, κ_{c_2}, each on one fresh instance
  update U_{c_1}, U_{c_2}, L_{c_1}, L_{c_2}
  for each $c \in C$ **if** doubling condition is met for c, **double** κ_c and rerun all capped runs

until interrupted
return $\arg\max_c L_c$ // configuration with largest lower bound

Continuous Optimistic Utilitarian Procrastination

COUP (Continuous Optimistic Utilitarian Procrastination) [Graham, Rojas & LB, AAAI 2026]

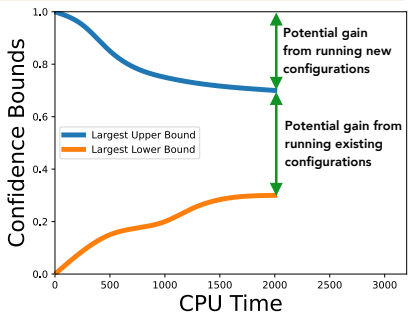
initialize C by sampling a small set of configurations
initialize $\kappa_c = 1$ for each configuration $c \in C$ // initial captime; choice of units up to the user
initialize $U_c = 1$ and $L_c = 0$ for each configuration $c \in C$ // initial bounds on expected utility from c
repeat
    select configuration $c_1 \in C$ having max average utility across all runs performed so far
    select configuration $c_2 \in \arg\max_{c \in C \setminus \{c_1\}} U_c$ // configuration other than c_1 with largest upper bound
    run c_1, c_2 with captimes κ_{c_1}, κ_{c_2}, each on one fresh instance
    update U_{c_1}, U_{c_2}, L_{c_1}, L_{c_2}
    for each $c \in C$ **if** doubling condition is met for c, **double** κ_c and rerun all capped runs
    if add condition is met:
        sample a new configuration c' and add it to C
until interrupted
return $\arg\max_c L_c$ // configuration with largest lower bound

Continuous Optimistic Utilitarian Procrastination

COUP (Continuous Optimistic Utilitarian Procrastination) [Graham, Rojas & LB, AAAI 2026]

initialize C by sampling a small set of configurations
initialize $\kappa_c = 1$ for each configuration $c \in C$ // initial captime; choice of units up to the user
initialize $U_c = 1$ and $L_c = 0$ for each configuration $c \in C$ // initial bounds on expected utility from c
repeat
  select configuration $c_1 \in C$ having max average utility across all runs performed so far
  select configuration $c_2 \in \arg\max_{c \in C \setminus \{c_1\}} U_c$ // configuration other than c_1 with largest upper bound
  run c_1, c_2 with captimes κ_{c_1}, κ_{c_2}, each on one fresh instance
  update U_{c_1}, U_{c_2}, L_{c_1}, L_{c_2}
  for each $c \in C$ **if** doubling condition is met for c, **double** κ_c a
  if add condition is met:
    sample a new configuration c' and add it to C
until interrupted
return $\arg\max_c L_c$ // configuration with largest lower bound

Add condition: More potential gain from new configs than from existing ones:



Theoretical Guarantee made by COUP

(ϵ, γ)-optimality

A configuration is (ϵ, γ)-optimal if its expected utility is no more than ϵ less than that of the best configuration that remains after excluding the top γ-fraction of all configurations.

As mentioned earlier, we're able to drop the β parameter bounding the fraction of capped runs.

Theorem [Graham & LB, ICLR 2025]

COUP returns an (ϵ, γ)-optimal configuration with probability at least $1 - \delta$, with $\epsilon = \max_c U_c - \max_{c'} L_{c'}$ and $\gamma = n \log \frac{2}{\delta}$, and where n denotes the total number of configurations sampled and δ is a given failure probability.

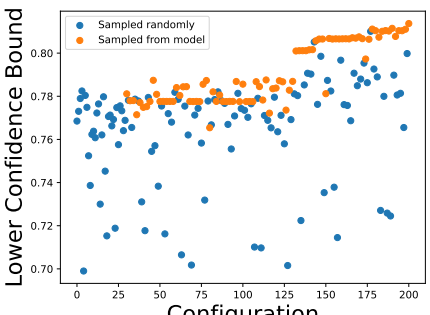
- ϵ and γ both shrink toward 0 as COUP runs
  - As COUP runs, it continually tightens the loosest upper and lower bounds; it grows n by sampling more configurations
  - We can choose how to trade off improvements in ϵ vs γ. Default: same speed.

COUP: Incorporating a Prediction Model

- As COUP runs we can **fit a predictive model** to performance data observed so far, just like SMAC does
- Half of configurations sampled randomly $\Rightarrow$ **all the same guarantees hold** up to a constant factor

COUP: Incorporating a Prediction Model

- As COUP runs we can **fit a predictive model** to performance data observed so far, just like SMAC does
- Half of configurations sampled randomly $\Rightarrow$ **all the same guarantees hold** up to a constant factor
- We use XGBoost to learn a forest of boosted trees
- Configurations sampled from the model were **consistently best**



Talk Overview

Algorithm Configuration

Algorithm Configuration with Theoretical Guarantees

Reconsidering the Objective Function

Utilitarian Algorithm Configuration

Empirical Evaluation

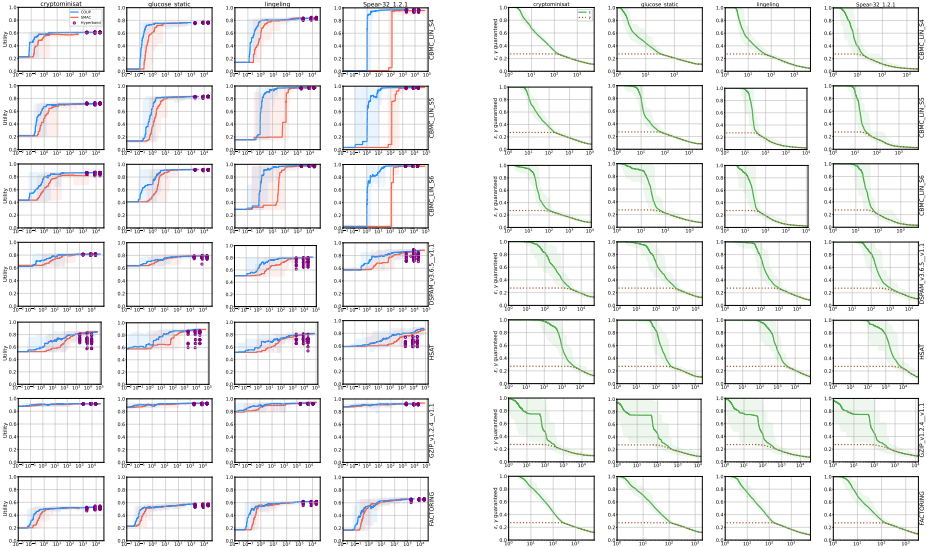
ACLib Experiments [Graham & KLB, working paper]

- **ACLib**, a library of algorithm configuration scenarios
  [Hutter, López-Ibánez, Fawcett, Lindauer, Hoos, KLB & Stützle, 2014]
- Compared COUP to
  - **SMAC3** [Lindauer, Eggensperger, Feurer, Biedenkapp, Deng, Benjamins, Ruhkopf, Sass & Hutter, 2022]
  - **Hyperband** [Li, Jamieson, DeSalvo, Rostamizadeh & Talwalkar, 2018]
- 4 **solvers**
- 20 **instance distributions**
- 3 **utility functions**
- Averaged over 10 **independent repetitions** to allow visualization of variance

ACLib Experiments - Choosing utility functions [Graham & KLB, working paper]

- In the utility-optimization case, many solver–distribution–utility function triples are **trivially easy to optimize**:
  - **all configurations are good**: $u(t) = 1$ for all $t < 24$ hours, but t is on the order of seconds for all configurations
  - **all configurations are bad**: $u(t) = 0$ for all $t > 1$ second, but t is on the order of hours for all configurations
  - key issue is not the choice of utility function but the **constants within the utility function**
- **Set utility function parameters** to values that lead to interesting computational problems for each solver–distribution pair, using a "pre-training" dataset:
  - **Exponential utility function**: $u(t) = \exp(-t/\kappa_0)$, with κ_0 chosen to be the pre-training sample mean
  - **Pareto utility function**: $u(t) = \left(\frac{\kappa_0}{t}\right)^a$, with κ_0 chosen to be an absolute minimum process time and a chosen so that the fastest run seen has utility 0.5
  - **Linear utility function**: $u(t) = 1 - \frac{t}{\kappa_0}$, with κ_0 chosen so that the best pre-training config has mean utility 0.5

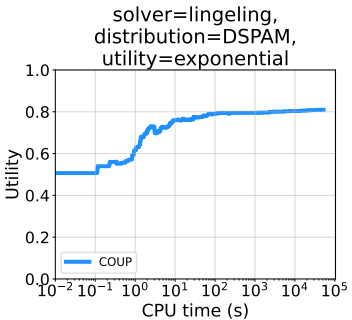
ACLib Experiments - Many scenarios... [Graham & KLB, working paper]



•••

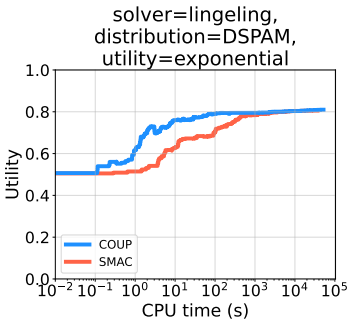
ACLib Experiments - Utility [Graham & KLB, working paper]

A typical example using an exponential utility function



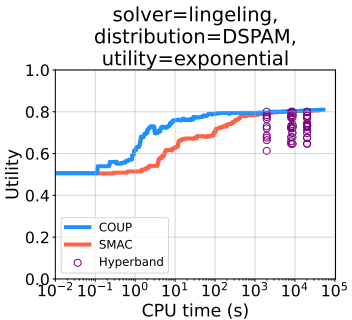
ACLib Experiments - Utility [Graham & KLB, working paper]

A typical example using an exponential utility function



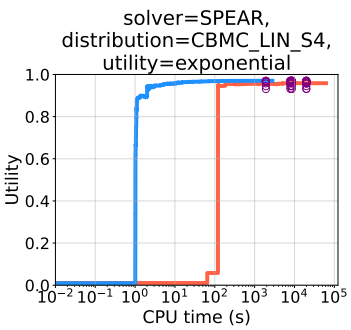
ACLib Experiments - Utility [Graham & KLB, working paper]

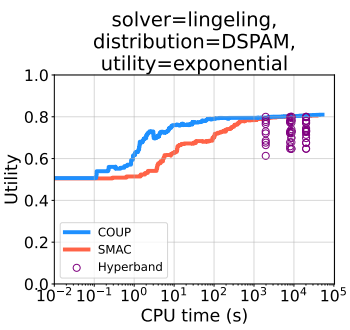
A typical example using an exponential utility function

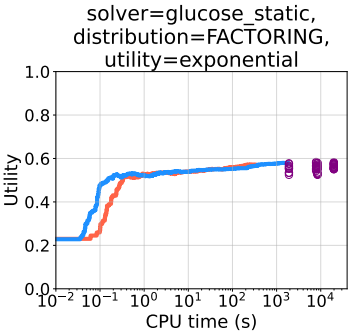


ACLib Experiments - Utility [Graham & KLB, working paper]

A typical example using an exponential utility function



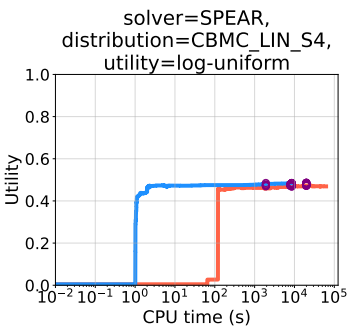


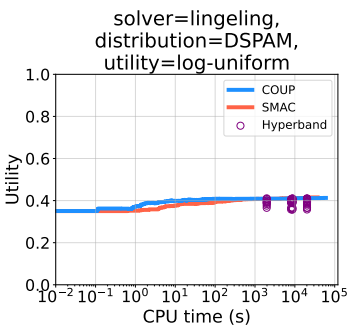


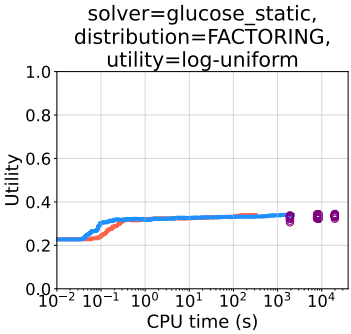
All procedures ultimately find good configurations,
but COUP typically stochastically dominates SMAC,
and outperforms Hyperband runs while also offering anytime performance.

ACLib Experiments - Utility [Graham & KLB, working paper]

And using a Pareto utility function



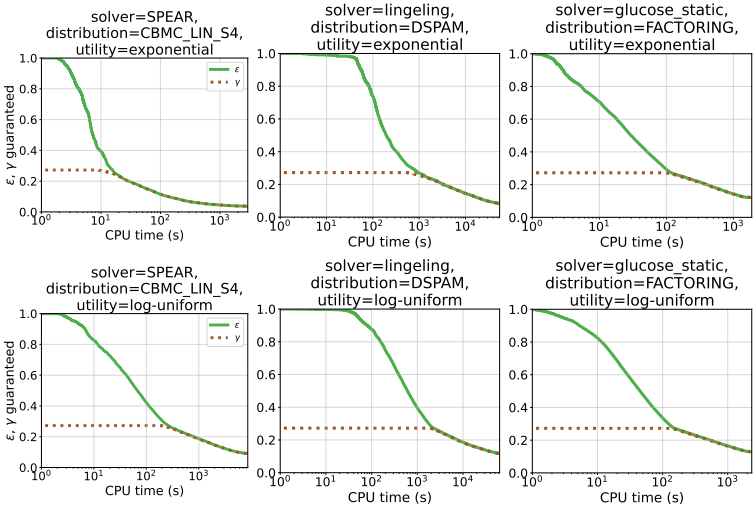




All procedures ultimately find good configurations,
but COUP typically stochastically dominates SMAC,
and matches/exceeds Hyperband runs in an anytime fashion.

ACLib Experiments - Theoretical Guarantees [Graham & KLB, working paper]

COUP also offers anytime optimality guarantees!



Conclusions

- **Algorithm configuration** learns algorithms that perform well on data
  - the key issue is being smart about evaluation budgets
  - that's why LLMs are not going to make it obsolete
  - though they might yield useful proposal distributions for new configurations
- To date, the most **practical AC techniques** have been heuristics like SMAC
- More recent (impractical) work performs AC in a **theoretically sound way**
- Regardless of how you do AC, you should use a **utilitarian objective**
- Utilitarian AC is **actually easier**, particularly for proving theoretical guarantees
- **COUP** is a state-of-the-art utilitarian configuration procedure:
  - stochastically dominates SMAC and you get guarantees for free
  - outperforms Hyperband and you get anytime performance for free

[height=6em]qr.png