Procrastinating with Confidence
Near-Optimal, Anytime, Adaptive Algorithm Configuration

Brendan Lucier, Microsoft Research

Bobby Kleinberg, Cornell
Kevin Leyton-Brown, UBC
Devon Graham, UBC

TTIC Workshop on Automated Algorithm Design, August 7 2019
Algorithm Configuration

Automating Algorithm Design
- Encode design choices as parameters
- Search for good configurations via learning
- Can tailor to specific contexts (input distribs)

Examples: solvers for SAT, MIPs, TSP instances, ...

Goal: find good configurations quickly
Good = fast algorithms for relevant problem instances

ParamILS [Hutter, Hoos, Leyton-Brown, Stützle 2009],
SMAC [Hutter, Hoos, Leyton-Brown 2011],
Hyperband [Li, Jamieson, DeSalvo, Rostamizadeh, Talwalkar 2016], ...
Algorithm Configuration

Two parts of the algorithm configuration problem:

1. Which configurations should we test?
   • Predict promising new configurations
   • Bayesian methods, structural assumptions, ...

2. How should we efficiently test configurations?
   • Test by running algorithms on random inputs
   • How many inputs to try on each configuration?
   • Goal: don’t waste time on duds

This work
This Work (informal):

**Structured Procrastination** [2017]: algorithm configuration procedure with guaranteed worst-case running time.
- Find approx. optimal config. in time proportional to \([\# \, \text{configs}] \times [\text{OPT running time}] \times [\text{error terms}]\).
- Nearly matches worst-case lower bounds (up to logs)

**Structured Procrastination with Confidence** [2019]:
Bounds can be made adaptive, better for “easier” instances
- See also: Leaps & Bounds, Caps & Runs
  [Weisz, György, Szepesvári 2018, 2019]

**Anytime procedures:** user stops search procedure at any point, guarantee tightens over time.
- User does not need to pre-specify error bounds
Model

Problem instance:

\( N \) – Collection of algorithm configurations
  • For now: assume \(|N| = n\) is small

\( \Gamma \) – Distribution over input instances

\( R(i, j) \) – Runtime of configuration \( i \) on input \( j \)

\[
R(i) = E_{j \sim \Gamma}[R(i, j)]
\]

\( \kappa_0 \) – Minimum runtime: \( R(i, j) \geq \kappa_0 > 0 \)

Can \textit{cap} runs at a \textit{timeout threshold} \( \theta \):

\[
R_\theta(i) = E_{j \sim \Gamma}[\min\{R(i, j), \theta\}]
\]
Model

\[ \text{OPT} = \min_i \{ R(i) \} \]

Configuration \( i \) is \( \epsilon \)-optimal if \( R(i) \leq (1 + \epsilon) \text{OPT} \)

Goal (?): find an \( \epsilon \)-optimal configuration

Example: \[ R(A) = \begin{cases} 1 & \text{w.p. } 1 - 10^{-20} \\ 10^{30} & \text{otherwise} \end{cases} \]

\[ R(B) = 1000 \]

Then \( R(B) \ll R(A) \), but two issues:

- Driven by rare but very bad inputs; user may prefer to cap
- Even if \( R(A) = 1 \) always, need to run \( 10^{20} \) tests to check!
Model

A relaxed objective:
Config. $i$ is $(\epsilon, \delta)$-optimal if there is a threshold $\theta$ such that

- $R_\theta(i) \leq (1 + \epsilon) \text{OPT}$
- $\text{Pr}_{j \sim \Gamma}[R(i, j) > \theta] \leq \delta$

Note: $(\epsilon, 0)$-optimal is equivalent to $\epsilon$-optimal.
Structured Procrastination

[Kleinberg, Leyton-Brown, L 2017], [Kleinberg, Leyton-Brown, L, Graham 2019]

Theorem:
There is an anytime procedure that, when terminated after $\tilde{\Omega}\left(OPT \cdot \frac{n}{\varepsilon^2 \delta}\right)$ steps, returns an $(\varepsilon, \delta)$-optimal configuration with high probability.

Notes:
- $\tilde{\Omega}$ suppresses log factors, including log of max running time; improved by [Weisz, György, Szepesvári 2018]
- Nearly tight: matching lower bound up to log factors
Toyz Example

2 configurations A and B
B has deterministic runtime \( \kappa \)
Decide if A is \((\epsilon, \delta)\)-optimal in time \( O(\kappa \cdot POLY(\epsilon, \delta)) \)

Idea #1: Run A on \( T = O(1/\epsilon^2 \delta) \) inputs \( \times \)
- estimate runtime of A excluding top \( \delta \) quantile
- Compare w/ \( \kappa \) to determine if A is \((\epsilon, \delta)\)-suboptimal

Bad example: A has deterministic runtime \( \gg \kappa \)
Toy Example

2 configurations A and B
B has deterministic runtime $\kappa$
Decide if A is $(\varepsilon, \delta)$-optimal in time $O(\kappa \cdot \text{POLY}(\varepsilon, \delta))$

Idea #1: Run A on $T = O(1/\varepsilon^2 \delta)$ inputs \textbf{X}
Idea #2: Run A on inputs for total time $O(\kappa/\varepsilon^2 \delta)$ \textbf{X}
• Estimate CDF of $R(A)$ from completed runs

Bad example: $R(A) = \begin{cases} 
\kappa/2 & \text{w.p. } 1 - 2\delta \\
\kappa/2\delta & \text{w.p. } \delta \\
\gg \kappa/\varepsilon^2 \delta & \text{w.p. } \delta
\end{cases}$

Hit bad input early ($\sim O(1/\delta)$ runs), waste all our time there
Toy Example

2 configurations A and B
B has deterministic runtime $\kappa$
Decide if A is $(\epsilon, \delta)$-optimal in time $O(\kappa \cdot POLY(\epsilon, \delta))$

Idea #1:

$Ideas #2:\quad R(A) = \begin{cases} 
\kappa/2 & \text{w.p. } 1 - 2\delta \\
\kappa/2\delta & \text{w.p. } \delta \\
\gg \kappa/\epsilon^2\delta & \text{w.p. } \delta 
\end{cases}$

Idea #3:

- Run A on $O(1/\epsilon^2\delta)$ inputs for total time $O(\kappa/\epsilon^2\delta)$, but...
- Set a captime for each run (e.g., $\kappa$)
- If hit cap, pause that run and move on to the next
- Only return to a run if $\delta$ fraction of runs are paused
Structured Procrastination

A time management scheme due to Stanford philosopher John Perry [2011 Ig Nobel prize, Literature]

• Keep a set of hard tasks that you procrastinate to avoid, thereby accomplish other tasks.
• Eventually replace each daunting task with a new task that is even more daunting, and so complete the former.

Structured Procrastination Algorithm Configuration:

• Maintain sets of tasks (for each config., a queue of runs)
• Start with the easiest tasks (shortest captimes)
• Procrastinate when these tasks prove daunting (put capped runs back on the queue)
Implementation

1. Initialize a bounded-length queue $Q_i$ of (input, captimes) pairs for each configuration $i$.
   - Instances randomly sampled from $\Gamma$
   - Initial captimes of $\kappa_0$
Implementation

1. Initialize a **bounded-length queue** $Q_i$ of (input, captime) pairs for each configuration $i$.

2. Calculate a **runtime estimate** for each configuration $i$
   - Optimistic empirical average runtime: treat any capped runs in the queue as if they finished at their captime
   - Initially $\kappa_0$ for new configurations
Implementation

1. Initialize a \textit{bounded-length queue} $Q_i$ of (input, captime) pairs for each configuration $i$.

2. Calculate a \textit{runtime estimate} for each configuration $i$.

3. Choose the configuration with \textit{fastest estimated runtime}, then select the (input, captime) pair from the head of its queue.
   - This will be the queue entry with smallest captime.
Implementation

1. Initialize a **bounded-length queue** $Q_i$ of (input, captime) pairs for each configuration $i$.

2. Calculate a **runtime estimate** for each configuration $i$.

3. Choose the configuration with **fastest estimated runtime**, then select the (input, captime) pair from the head of its queue.

4. If the task completes, generate a **new input** and add it to the queue.

5. Otherwise, **procrastinate**: double the captime and add the task back at the tail of the queue.
   - We will do many other runs before coming back to this task.
Implementation

1. Initialize a **bounded-length queue** $Q_i$ of (input, captime) pairs for each configuration $i$.

2. Calculate a **runtime estimate** for each configuration $i$.

3. Choose the configuration with **fastest estimated runtime**, then select the (input, captime) pair from the head of its queue.

4. If the task completes, generate a **new input** and add it to the queue.

5. Otherwise, **procrastinate**: double the captime and add the task back to the tail of the queue.

6. If execution hasn’t been interrupted yet, goto 2.

7. Return the configuration **we spent the most time running**
   - More statistically stable than return config. with best current estimate.
Implementation

1. Initialize a bounded-length queue $Q_i$ of (input, captime) pairs for each configuration $i$.
2. Calculate a runtime estimate for each configuration $i$.
3. Choose the configuration with fastest estimated runtime, then select the (input, captime) pair from the head of its queue.
4. If the task completes, generate a new input and add it to the queue.
5. Otherwise, procrastinate: double the captime and add the task back to the tail of the queue.
6. If execution hasn’t been interrupted yet, goto 2.
7. Return the configuration we spent the most time running.
   - More statistically stable than return config. with best current estimate.

Note: $\epsilon$ and $\delta$ only affect queue length.
Implementation (Anytime)

1. Initialize a bounded-length queue $Q_i$ of (input, captime) pairs for each configuration $i$.
2. Calculate a runtime estimate for each configuration $i$.
3. Choose the configuration with fastest estimated runtime, then select the (input, captime) pair from the head of its queue.
4. If the task completes, generate a new input and add it to the queue.
5. Otherwise, procrastinate: double the captime and add the task back to the tail of the queue.

5.5. Grow the chosen configuration’s queue (if necessary)

6. If execution hasn’t been interrupted yet, goto 2.

7. Return the configuration we spent the most time running.
   - More statistically stable than return config. with best current estimate.
Theorem: If the Structured Procrastination procedure is terminated after \( \tilde{\Omega} \left( OPT \cdot \frac{n}{\epsilon^2 \delta} \right) \) steps, it returns an \((\epsilon, \delta)\)-optimal configuration with high probability (in # of steps).
Performance Guarantee

Theorem: If the Structured Procrastination procedure is terminated after $\widetilde{\Omega}\left(OPT \cdot \frac{n}{\epsilon^2 \delta}\right)$ steps, it returns an $(\epsilon, \delta)$-optimal configuration with high probability (in # of steps).
**Theorem:** If the Structured Procrastination procedure is terminated after $\tilde{O} \left( OPT \cdot \frac{n}{\epsilon^2 \delta} \right)$ steps, it returns an $(\epsilon, \delta)$-optimal configuration with high probability (in # of steps).
Performance Guarantee

**Theorem:** If the Structured Procrastination procedure is terminated after $\tilde{\Omega} \left( OPT \cdot \frac{n}{\epsilon^2 \delta} \right)$ steps, it returns an $(\epsilon, \delta)$-optimal configuration with high probability (in # of steps).

**Lower Bound:** Suppose an algorithm configuration procedure is guaranteed to select $(\epsilon, \delta)$-optimal configuration with probability at least $\frac{1}{2}$. Then its worst-case expected running time must be at least $\Omega \left( OPT \cdot \frac{n}{\epsilon^2 \delta} \right)$. 
Lower Bound: Suppose an algorithm configuration procedure is guaranteed to select \((\epsilon, \delta)\)-optimal configuration with probability at least \(\frac{1}{2}\). Then its worst-case expected running time must be at least \(\Omega \left( OPT \cdot \frac{n}{\epsilon^2 \delta} \right) \).

\[
R(A) = \begin{cases} 
1 & \text{w.p. } 1 - 2\delta \\
1/\delta & \text{w.p. } 2\delta 
\end{cases} \quad R(B) = \begin{cases} 
1 & \text{w.p. } 1 - 2\delta(1 - \epsilon) \\
1/\delta & \text{w.p. } 2\delta(1 - \epsilon) 
\end{cases}
\]

- Instance: \(n - 1\) copies of A, 1 copy of B
- A is \((\epsilon, \delta)\)-suboptimal; procedure must return B
- Takes \(1/\epsilon^2 \delta\) runs to distinguish types A and B
- Need to check \(O(n)\) configs to find a B
Beating the Lower Bound

**Question:** Can we do better on “easier” instances?

**LeapsAndBounds, CapsAndRuns** [Weisz, György, Szepesvári 2018,2019] Improved performance on practical instances; require users to specify $\epsilon$ and $\delta$ (not anytime). **See next talk!**

**Structured Procrastination with Confidence (SPC):**

- Maintain **confidence bounds** on each config’s runtime
- **Bandits:** optimism in the face of uncertainty [Auer, Cosa Bianchi, Fischer 2002], [Bubeck, Cesa Bianchi 2012]
- Detect “obviously bad” configurations more quickly
- Running time matches (up to log factors) the running time of a hypothetical “optimality verification procedure” that knows each configuration’s runtime distribution
Structured Procrastination with Confidence

1. Initialize a bounded-length queue \( Q_i \) of (input, captime) pairs for each configuration \( i \).

2. Calculate a runtime estimate for each configuration \( i \) with lowest confidence bound.

3. Choose the configuration with fastest estimated runtime, then select the (input, captime) pair from the head of its queue.

4. If the task completes, generate a new input and add it to the queue.

5. Otherwise, procrastinate: double the captime and add the task back to the tail of the queue.

5.5. Grow the chosen configuration’s queue (if necessary)

6. If execution hasn’t been interrupted yet, goto 2.

7. Return the configuration we spent the most time running.
Details: Confidence Bounds

Idea: adjust empirical CDF non-uniformly; get lower bound $L_i$.
Construction: empirical process theory [Wellner ’78]

Key Lemma: if configuration $i$ is $(\epsilon, \delta)$-suboptimal, then after $\tilde{O}(1/\epsilon^2 \delta)$ executions we will have $L_i > \text{OPT}$.

I.e., we expect to run configuration $i$ at most $\tilde{O}(1/\epsilon^2 \delta)$ times.
Analysis

Key Lemma: if configuration $i$ is $(\epsilon, \delta)$-suboptimal, then after $\tilde{O}(1/\epsilon^2 \delta)$ executions we will have $L_i > \text{OPT}$.

Note: can apply different $(\epsilon, \delta)$ pairs to each config!

Example:

- Config A is optimal
- Config B is $(\frac{1}{10}, \frac{1}{100})$-suboptimal
- Config C is $(\frac{1}{2}, \frac{1}{2})$-suboptimal

- C is “easier” to exclude; can be quickly verified suboptimal
- SPC will run configuration C fewer times
Performance Guarantee

For any \( \epsilon \) and \( \delta \), and each configuration \( i \), define

\[
V_i(\epsilon, \delta) = \begin{cases} 
1/\epsilon^2\delta & \text{if } i \text{ is } (\epsilon, \delta)\text{-optimal} \\
\min_{\tilde{\epsilon}, \tilde{\delta}: i \text{ is } (\tilde{\epsilon}, \tilde{\delta})\text{-suboptimal}} \{1/\tilde{\epsilon}^2\tilde{\delta}\} & \text{otherwise}
\end{cases}
\]

**Intuition:** \( V_i(\epsilon, \delta) \) is min. \# runs that an omniscient verifier needs to convince a skeptic that \( i \) is/Isn't \((\epsilon, \delta)\)-optimal.

**Theorem:** If Structured Procrastination with Confidence is terminated after \( \tilde{\Omega}(OPT \cdot \sum_{i \in N} V_i(\epsilon, \delta)) \) steps, it returns an \((\epsilon, \delta)\)-optimal configuration with high probability.
Evaluation (I)

Are practical instances “easy” (variation in suboptimality)?

Publicly-available data from [Hutter Xu Hoos Leyton-Brown 2014]: SPEAR SAT solver, SWV problem instances [Babić, Hu 2007].
Evaluation (II)

Does this result in faster practical performance?
Data from [Weisz, György, Szepesvári 2018]:
• 972 minisat configurations
• 20118 nontrivial CNFuzzDD SAT instances
• Fix $\epsilon = 0.1$, track time to find $(\epsilon, \delta)$-optimal configuration

Proof Technique: simulate execution time as if we had run using Structured Procrastination, to obtain $(\epsilon, \delta)$ guarantee
Extension: many configurations

So far, we’ve assumed $|N| = n$ is small.

Typical case: N is very large (or infinite); $\Omega(n)$ is infeasible

Relaxed Benchmark: a config. within the top $\gamma$-performing quantile, over all configurations in $N$.

$\text{OPT}_\gamma$: $\gamma$ fraction of configurations have $R(i) < \text{OPT}_\gamma$

Config. $i$ is $(\epsilon, \delta, \gamma)$-optimal if there is a threshold $\theta$ such that

- $R_\theta(i) \leq (1 + \epsilon)\text{OPT}_\gamma$
- $\Pr_{j \sim \Gamma}[R(i, j) > \theta] \leq \delta$
Extension: many configurations

Config. $i$ is $(\epsilon, \delta, \gamma)$-optimal if there is a threshold $\theta$ such that
- $R_\theta(i) \leq (1 + \epsilon)\text{OPT}_\gamma$
- $\mathbf{Pr}_{j \sim \Gamma}[R(i, j) > \theta] \leq \delta$

**Idea 1:** Sample $O(1/\gamma)$ configurations from $N$, then run SPC on the resulting set of configurations.
- Best sampled configuration is likely to have $R(i) < \text{OPT}_\gamma$

**Idea 2:** Gradually increase the number of configurations in the sample, as SPC runs.
- Leads to an anytime guarantee with respect to $\gamma$
Extension: many configurations

**Theorem:** If the Structured Procrastination procedure is terminated after $\widetilde{\Omega}\left(\text{OPT}_\gamma \cdot \frac{1}{\epsilon^2 \delta\gamma}\right)$ steps, it identifies an $(\epsilon, \delta, \gamma)$-optimal configuration with high prob. (in # of steps).

**Lower Bound:** Suppose an algorithm configuration procedure is guaranteed to select $(\epsilon, \delta, \gamma)$-optimal configuration with probability at least $\frac{1}{2}$. Then its worst-case expected running time must be at least $\Omega\left(\text{OPT}_\gamma \cdot \frac{1}{\epsilon^2 \delta\gamma}\right)$.

**Note:** a corresponding result for SPC; replace $1/\epsilon^2 \delta$ with [expected time to verify suboptimality of random config].
Summary

**Structured Procrastination**: approach to algorithm configuration.

- Procrastinates on potentially hard inputs rather than solving them to completion when first encountered

**Anytime procedure**, guaranteed to find an approx. optimal algorithm configuration in nearly optimal worst-case time.

**Extension**: adaptively better performance on “easy” instances

- E.g., presence of bad configurations that can be rejected quickly

**Future directions:**

- Combining with Bayesian optimization, other methods
- Thorough empirical evaluations, comparisons

Thanks!