Procrastinating with Confidence

Near-Optimal, Anytime, Adaptive Algorithm Configuration

Brendan Lucier, Microsoft Research

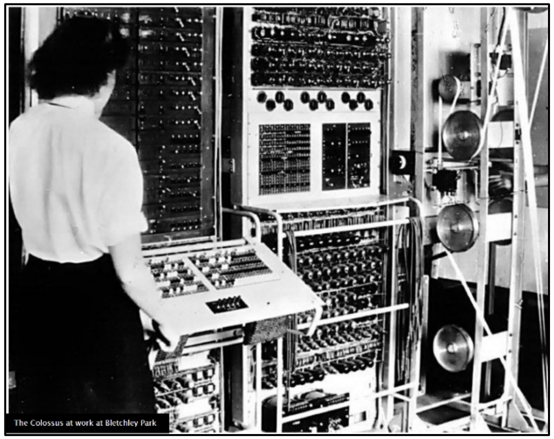
Bobby Kleinberg, Cornell

Kevin Leyton-Brown, UBC

Devon Graham, UBC

TTIC Workshop on Automated Algorithm Design, August 7 2019

Algorithm Configuration


The Colossus at work at Bletchley Park

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 [# configs] x [OPT running time] x [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

Γ – Distribution over input instances

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

$$R(i) = \mathbf{E}_{j\sim\Gamma}[R(i,j)]$$

κ_0 – Minimum runtime: $R(i,j) \geq \kappa_0 > 0$

Can *cap* runs at a timeout threshold θ:

$$R_\theta(i) = \mathbf{E}_{j\sim\Gamma}[\min\{R(i,j), \theta\}]$$

Model

$\text{OPT} = \min_i\{R(i)\}$

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

Goal (?): find an ϵ-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!

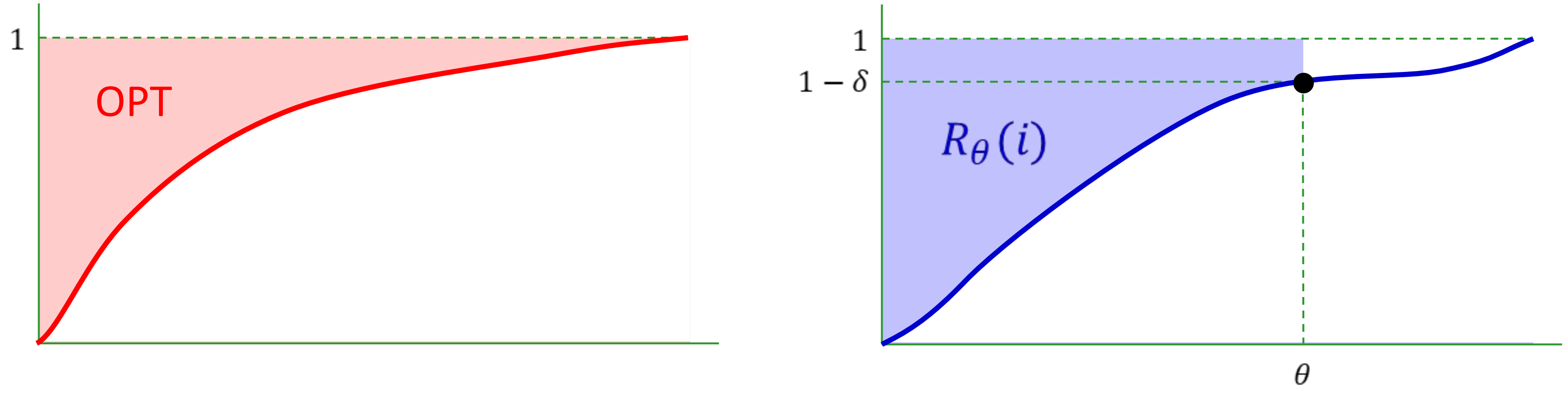
Model

A relaxed objective:

Config. i is (ϵ, δ)-optimal if there is a threshold θ such that

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

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



Structured Procrastination

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

Theorem:

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

Notes:

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

Toy Example

2 configurations A and B

B has deterministic runtime κ

Decide if A is (ϵ, δ)-optimal in time $O(\kappa \cdot POLY(\epsilon, \delta))$

Idea #1: Run A on $T = O(1/\epsilon^2\delta)$ inputs ✗

- estimate runtime of A excluding top δ quantile
- Compare w/ κ to determine if A is (ϵ, δ)-suboptimal

Bad example: A has deterministic runtime $\gg \kappa$

Toy Example

2 configurations A and B

B has deterministic runtime κ

Decide if A is (ϵ, δ)-optimal in time $O(\kappa \cdot POLY(\epsilon, \delta))$

Idea #1: Run A on $T = O(1/\epsilon^2\delta)$ inputs ✗

Idea #2: Run A on inputs for total time $O(\kappa/\epsilon^2\delta)$ ✗

- Estimate CDF of $R(A)$ from completed runs

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

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

Toy Example

2 configurations A and B

B has deterministic runtime κ

Decide if A is (ϵ, δ)-optimal in time $O(\kappa \cdot POLY(\epsilon, \delta))$

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

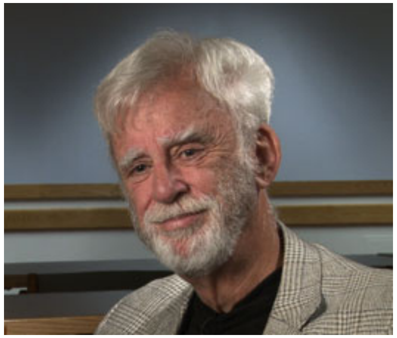
Idea #1:

Idea #2: … $/\epsilon^2\delta)$ ✗

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., κ)
- If hit cap, pause that run and move on to the next ✓
- Only return to a run if δ 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 accomplishing 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)

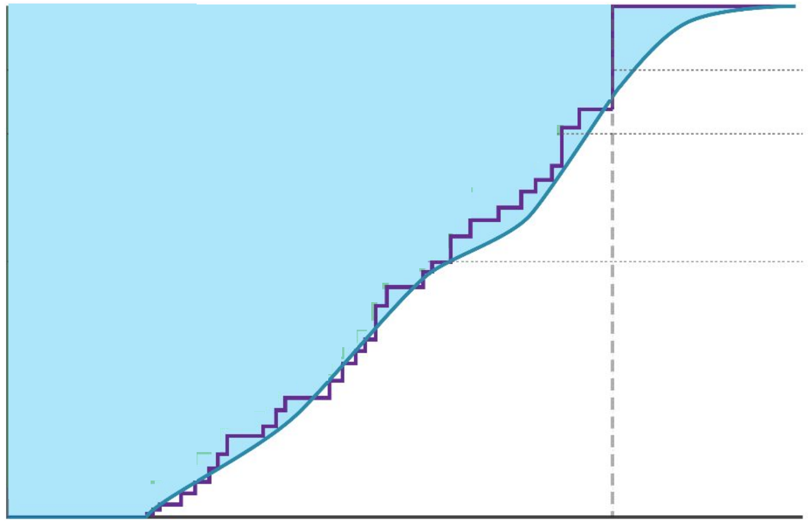
Implementation

1. Initialize a bounded-length queue Q_i of (input, captime) pairs for each configuration i.
    - Instances randomly sampled from Γ
    - Initial captimes of κ_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 κ_0 for new configurations



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
   - 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

Note: ϵ and δ only affect queue length

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 (Anytime)

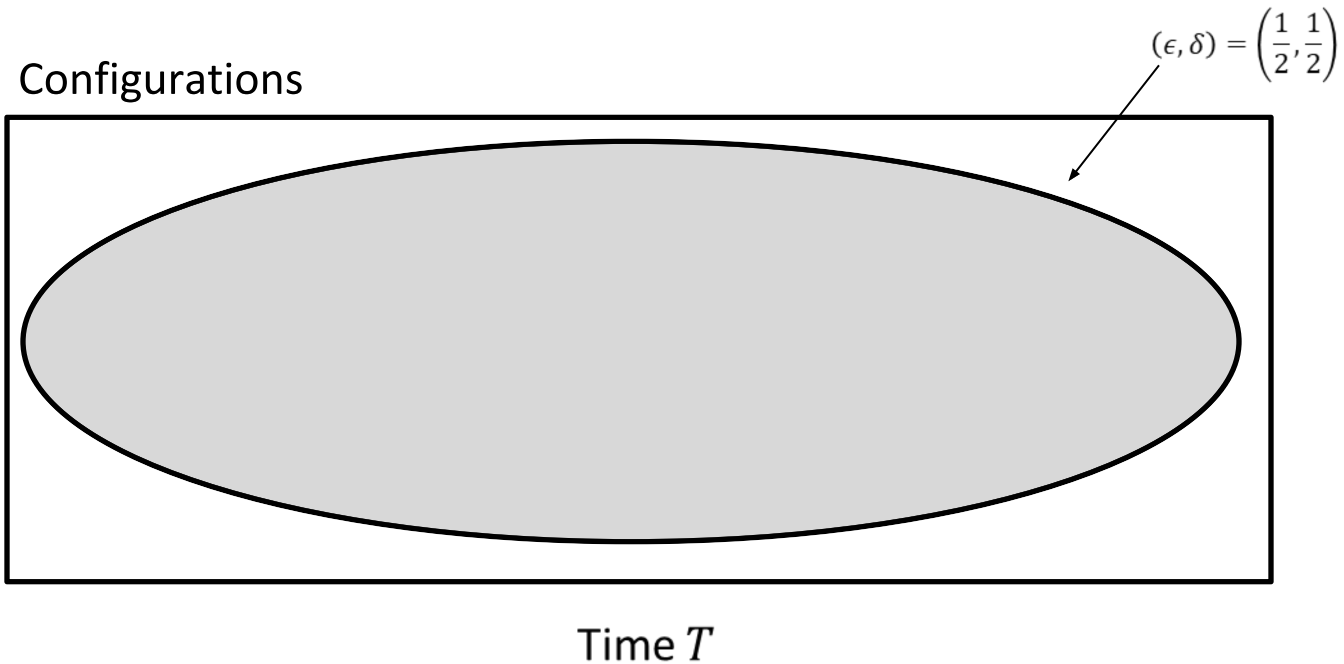
1. Initialize a bounded-length queue Q_i of (input, captime) pairs for each configuration i.
2. Calculate a runtime estimate for each configuration i
3. Choose the configuration with fastest estimated runtime, then select the (input, captime) pair from the head of its queue
4. If the task completes, 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

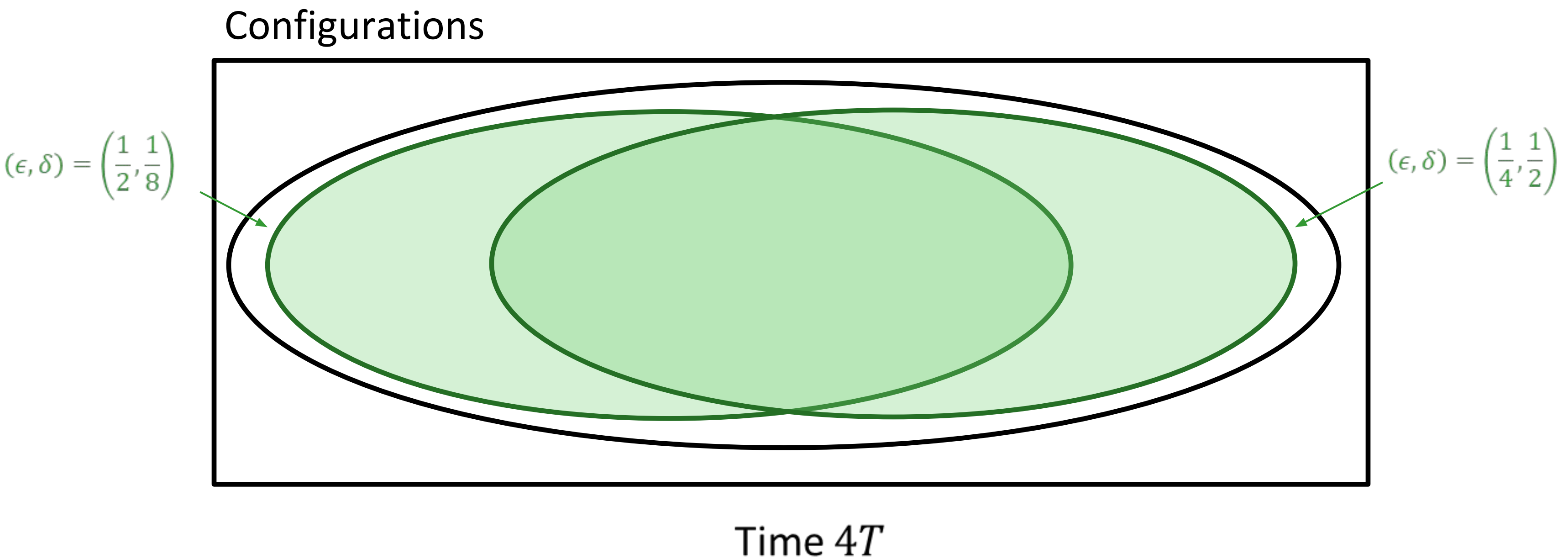
Performance Guarantee

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



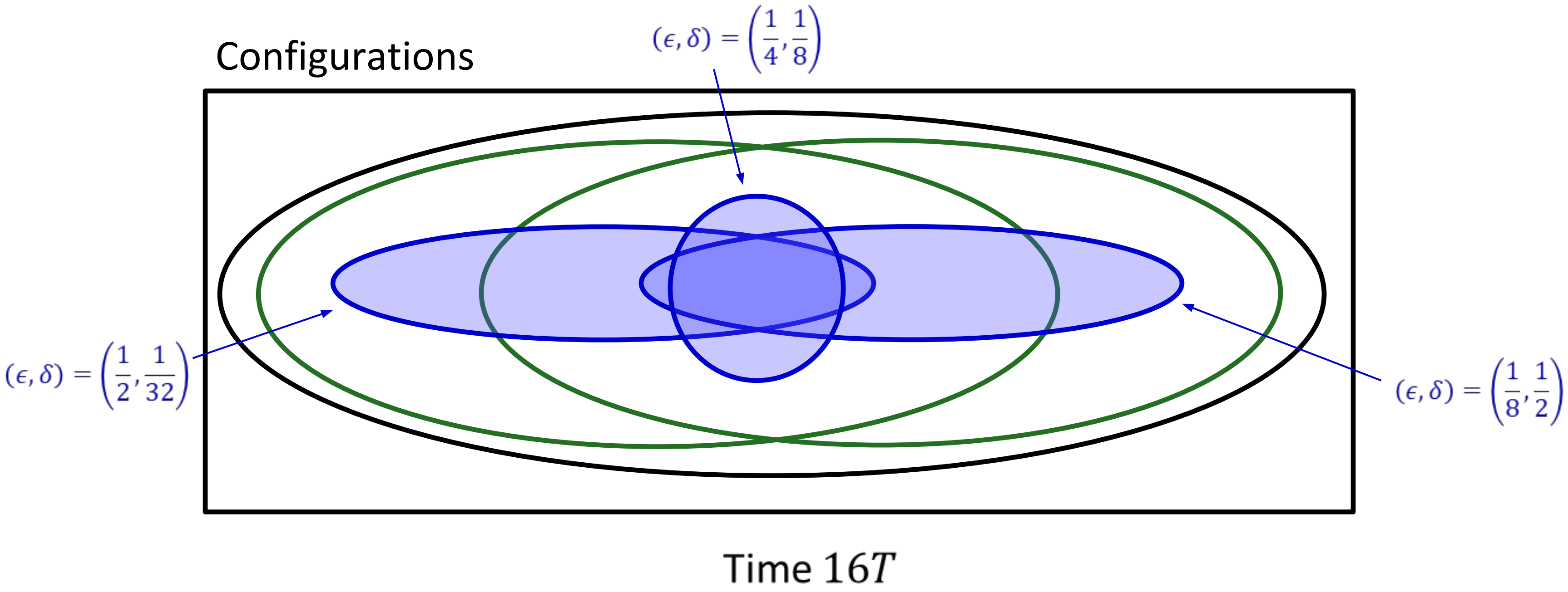
Performance Guarantee

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

Lower Bound: Suppose an algorithm configuration procedure is guaranteed to select (ϵ, δ)-optimal configuration with probability at least ½. Then its worst-case expected running time must be at least $\Omega\left(OPT \cdot \frac{n}{\epsilon^2 \delta}\right)$.

Lower Bound

Lower Bound: Suppose an algorithm configuration procedure is guaranteed to select (ϵ, δ)-optimal configuration with probability at least ½. Then its worst-case expected running time must be at least $\Omega\left(OPT \cdot \frac{n}{\epsilon^2 \delta}\right)$.

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

- Instance: $n-1$ copies of A, 1 copy of B
- A is (ϵ, δ)-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 ϵ and δ (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

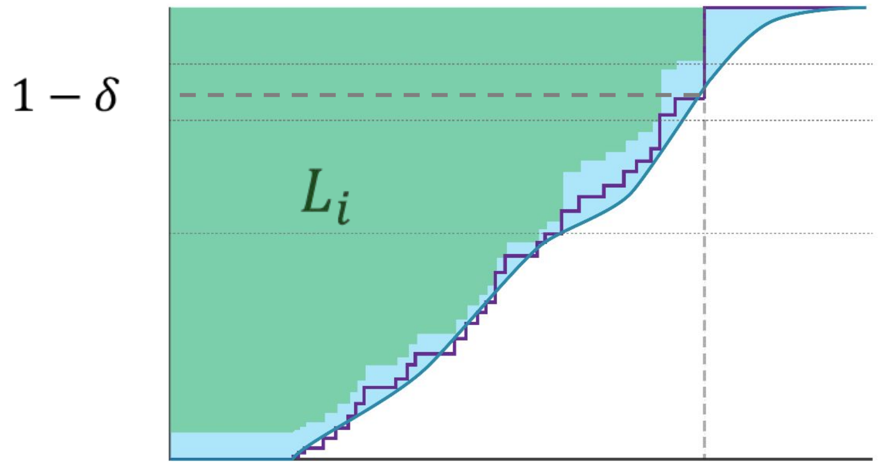
Structured Procrastination with Confidence

1. Initialize a bounded-length queue Q_i of (input, captime) pairs for each configuration i.
2. Calculate a ~~runtime estimate~~ lower confidence bound for each configuration i
3. Choose the configuration with ~~fastest estimated runtime~~ lowest confidence bound, 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~~ ran most often

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 (ϵ, δ)-suboptimal, then after $\tilde{O}(1/\epsilon^2\delta)$ executions we will have $L_i > \mathrm{OPT}$.

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

Analysis

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

Note: can apply different (ϵ, δ) pairs to each config!

Example:

Config A is optimal

Config B is $({}^1/_{10}, {}^1/_{100})$-suboptimal

Config C is $({}^1/_2, {}^1/_2)$-suboptimal

- C is "easier" to exclude; can be quickly verified suboptimal
- SPC will run configuration C fewer times

Performance Guarantee

For any ϵ and δ, and each configuration i, define

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

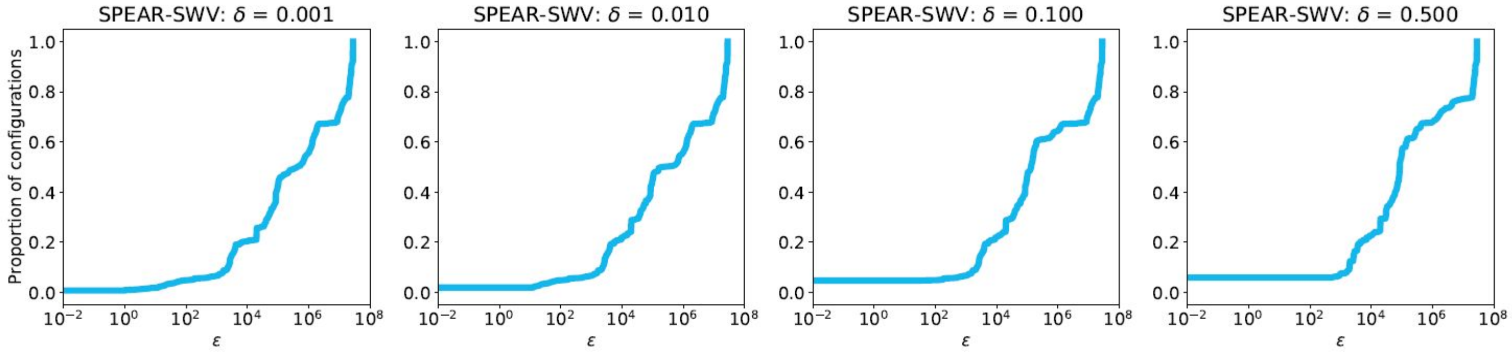
Intuition: $V_i(\epsilon, \delta)$ is min. # runs that an omniscient verifier needs to convince a skeptic that i is/isn't (ϵ, δ)-optimal.

Theorem: If Structured Procrastination with Confidence is terminated after $\widetilde{\Omega}(OPT \cdot \sum_{i \in N} V_i(\epsilon, \delta))$ steps, it returns an (ϵ, δ)-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].



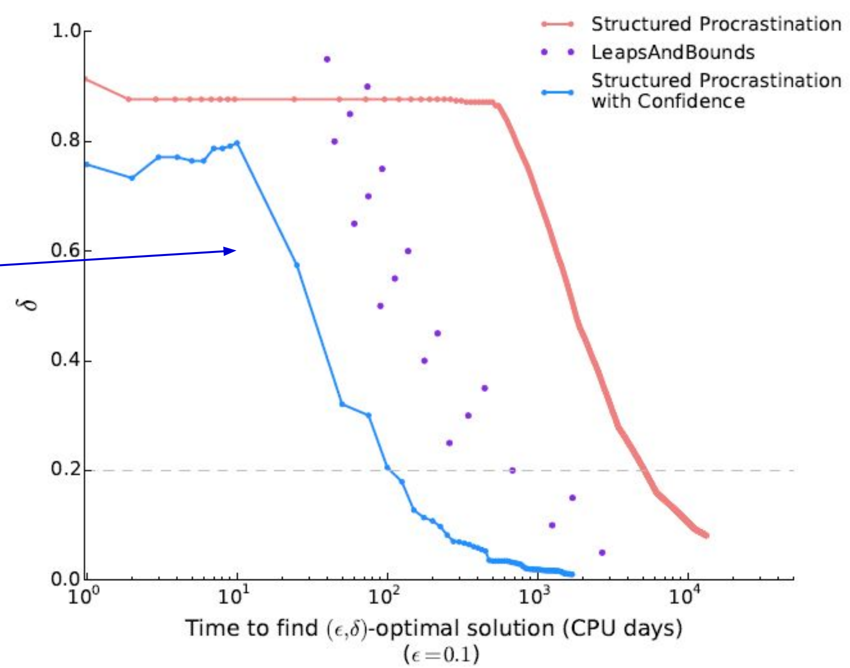
Evaluation (II)

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 (ϵ, δ)-optimal configuration

Proof Technique: simulate execution time as if we had run using Structured Procrastination, to obtain (ϵ, δ) 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 γ-performing quantile, over all configurations in N.
OPT_γ: γ fraction of configurations have $R(i) < \mathrm{OPT}_\gamma$

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

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

Extension: many configurations

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

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

Idea 1: Sample $\mathrm{O}(1/\gamma)$ configurations from N, then run SPC on the resulting set of configurations.

- Best sampled configuration is likely to have $R(i) < \mathrm{OPT}_\gamma$

Idea 2: Gradually increase the number of configurations in the sample, as SPC runs.

- Leads to an anytime guarantee with respect to γ

Extension: many configurations

Theorem: If the Structured Procrastination procedure is terminated after $\widetilde{\Omega}\left(OPT_\gamma \cdot \frac{1}{\epsilon^2 \delta \gamma}\right)$ 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 ½. Then its worst-case expected running time must be at least $\Omega\left(OPT_\gamma \cdot \frac{1}{\epsilon^2 \delta \gamma}\right)$.

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!