A Batch Sequential Halving Algorithm without Performance Degradation

Sotetsu Koyamada
koyamada@atr.jp
ATR, Kyoto University

Soichiro Nishimori
nishimori@ms.k.u-tokyo.ac.jp
The University of Tokyo

Shin Ishii
ishii@i.kyoto-u.ac.jp
Kyoto University, ATR

Abstract

In this paper, we investigate the problem of pure exploration in the context of multi-armed bandits, with a specific focus on scenarios where arms are pulled in fixed-size batches. Batching has been shown to enhance computational efficiency, but it can potentially lead to a degradation compared to the original sequential algorithm's performance due to delayed feedback and reduced adaptability. We introduce a simple batch version of the Sequential Halving (SH) algorithm (Karnin et al., 2013) and provide theoretical evidence that batching does not degrade the performance of the original algorithm under practical conditions. Furthermore, we empirically validate our claim through experiments, demonstrating the robust nature of the SH algorithm in fixed-size batch settings.

1 Introduction

In this study, we consider the pure exploration problem in the field of stochastic multi-armed bandits, which aims to identify the best arm within a given budget (Audibert et al., 2010). Specifically, we concentrate on the *fixed-size batch pulls* setting, where we pull a fixed number of arms simultaneously. Batch computation plays a crucial role in improving computational efficiency, especially in large-scale bandit applications where reward computation can be expensive. For instance, consider applying this to tree search algorithms like Monte Carlo tree search (Tolpin & Shimony, 2012). The reward computation here typically involves the value network evaluation (Silver et al., 2016; 2017), which can be computationally expensive. By leveraging batch computation and hardware accelerators (e.g., GPUs), we can significantly reduce the computational cost of the reward computation. However, while batch computation enhances computational efficiency, its performance (e.g., simple regret) may not match that of sequential computation with the same total budget, due to delayed feedback reducing adaptability. Therefore, the objective of this study is to develop a pure exploration algorithm that maintains its performance regardless of the batch size.

We focus on the *Sequential Halving* (SH) algorithm (Karnin et al., 2013), a popular and well-analyzed pure exploration algorithm. Due to its simplicity, efficiency, and lack of task-dependent hyperparameters, SH finds practical applications in, but not limited to, hyperparameter tuning (Jamieson & Talwalkar, 2016), recommendation systems (Aziz et al., 2022), and state-of-the-art AlphaZero (Silver et al., 2018) and MuZero (Schrittwieser et al., 2020) family (Danihelka et al., 2022). In this study, we aim to extend SH to a batched version that matches the original SH algorithm's performance, even with large batch sizes. To date, Jun et al. (2016) introduced a simple batched extension of SH and reported that it performed well in their experiments. However, the theoretical properties of batched SH have not yet been well-studied in the setting of fixed-size batch pulls.

We consider two simple and natural batched variants of SH (Sec. 3): *Breadth-first Sequential Halving* (BSH) and *Advance-first Sequential Halving* (ASH). We introduce BSH as an intermediate step to understanding ASH, which is our main focus. Our main contribution is providing a theoretical guarantee for ASH (Sec. 4), showing that *it is algorithmically equivalent to SH as long as the batch*

Algorithm 1 SH: Sequential Halving (Karnin et al., 2013)

input number of arms: n, budget: T
initialize best arm candidates $\mathcal{S}_0 \coloneqq [n]$
for round $r = 0, \dots, \lceil \log_2 n \rceil - 1$ **do**
  pull each arm $a \in S_r$ for $J_r = \left\lfloor \frac{T}{|\mathcal{S}_r| \lceil \log_2 n \rceil} \right\rfloor$ times
  $\mathcal{S}_{r+1} \leftarrow$ top-$\lceil |\mathcal{S}_r|/2 \rceil$ arms in $\mathcal{S}_r$ w.r.t. the empirical rewards
return the only arm in $\mathcal{S}_{\lceil \log_2 n \rceil}$

budget is not extremely small — For example, in a 32-armed stochastic bandit problem, ASH can match SH's choice with 100K sequential pulls using just 20 batch pulls, each of size 5K. This means that ASH can achieve the same performance as SH with significantly fewer pulls when the batch size is reasonably large. Moreover, one can understand the theoretical properties of ASH using the theoretical properties of SH, which have been well-studied (Karnin et al., 2013; Zhao et al., 2023). In our experiments, we validate our claim by comparing the behavior of ASH and SH (Sec. 5.1) and analyze the behavior of ASH with the extremely small batch budget as well (Sec. 5.2).

2 Preliminary

Pure Exploration Problem. Consider a pure exploration problem involving n arms and a budget T. We define a reward matrix $\mathcal{R} \in [0,1]^{n \times T}$, where each element $\mathcal{R}_{i,j} \in [0,1]$ represents the reward of the j-th pull of arm $i \in [n] \coloneqq \{1, \dots, n\}$, with j being counted independently for each arm. Each element in the i-th row is an i.i.d. sample from an unknown reward distribution of i-th arm with mean μ_i. Without loss of generality, we assume that $1 \geq \mu_1 \geq \mu_2 \geq \dots \geq \mu_n \geq 0$. In the standard sequential setting, a pure exploration algorithm sequentially observes T elements from $\mathcal{R}$ by pulling arms one by one for T times. The algorithm then selects one arm as the best arm candidate. Note that we only consider deterministic pure exploration algorithms in this study. Such an algorithm can be characterized by a mapping $\pi : [0,1]^{n \times T} \to [n]$ that takes $\mathcal{R}$ as input and outputs the selected arm a_T. The natural performance measure in pure exploration is the *simple regret*, defined as $\mathbb{E}_{\mathcal{R}}[\mu_1 - \mu_{a_T}]$ (Bubeck et al., 2009), which compares the performance of the selected arm a_T with the best arm 1.

Sequential Halving (SH; Karnin et al. (2013)) is a sequential elimination algorithm designed for the pure exploration problem. It begins by initializing the set of best arm candidates as $\mathcal{S}_0 \coloneqq [n]$. In each of the $\lceil \log_2 n \rceil$ rounds, the algorithm halves the set of candidates (i.e., $|\mathcal{S}_{r+1}| = \lceil |\mathcal{S}_r|/2 \rceil$) until it narrows down the candidates to a single arm in $\mathcal{S}_{\lceil \log_2 n \rceil}$. During each round $r \in \{0, \dots, \lceil \log_2 n \rceil - 1\}$, the arms in the active arm set $\mathcal{S}_r$ are pulled equally $J_r \coloneqq \left\lfloor \frac{T}{|\mathcal{S}_r| \lceil \log_2 n \rceil} \right\rfloor$ times, and the total budget consumed for round r is $T_r \coloneqq J_r \times |\mathcal{S}_r|$. The SH algorithm is described in Algorithm 1. We denote the mapping induced by the SH algorithm as π_{SH}. It has been shown that the simple regret of SH satisfies $\mathbb{E}_{\mathcal{R}}[\mu_1 - \mu_{a_T}] \leq \tilde{\mathcal{O}}(\sqrt{n/T})$, where $\tilde{\mathcal{O}}(\cdot)$ ignores the logarithmic factors of n (Zhao et al., 2023). Note that the consumed budget $\sum_{r < \lceil \log_2 n \rceil} T_r$ might be less than T. In this study, we assume that the remaining budget is consumed equally by the last two arms in the final round.

3 Batch Sequential Halving Algorithms

In this study, we consider the fixed-size batch pulls setting, where we simultaneously pull b arms for B times, with b being the fixed batch size and B being the batch budget (Jun et al., 2016). The standard sequential case corresponds to $b = 1$ and $B = T$. Our interest is to compare the performance of the batch SH algorithms with a large batch size b and a small batch budget B to that of the standard SH algorithm when pulling sequentially T times. Therefore, we compare the performance of the batch SH algorithms under the assumption that $T = b \times B$ holds, so that the total budget is the same in both the sequential and batch settings. In this section, we first reconstruct

Algorithm 2 SH (Karnin et al., 2013) implementation with target pulls $L^{\mathbf{B}}/L^{\mathbf{A}}$

input number of arms: n, budget: T
initialize empirical mean $\bar{\mu}_a \coloneqq 0$ and arm pulls $N_a \coloneqq 0$ for all $a \in [n]$
for $t = 0, \dots, T-1$ **do**
  let $\mathcal{A}_t$ be $\{a \in [n] \mid N_a = L_t\}$ ▷ L_t is either $L_t^{\mathbf{B}}$ (1) or $L_t^{\mathbf{A}}$ (2)
  pull arm $a_t \coloneqq \mathrm{argmax}_{a \in \mathcal{A}_t} \bar{\mu}_a$
  update $\bar{\mu}_{a_t}$ and $N_{a_t} \leftarrow N_{a_t} + 1$
return $\mathrm{argmax}_{a \in [n]}(N_a, \bar{\mu}_a)$

Algorithm 3 *Breadth-first* target pulls $L^{\mathbf{B}}$

input number of arms: n, budget: T
initialize empty $L^{\mathbf{B}}$, $K \coloneqq n$, $J \coloneqq 0$
for $r = 0, \dots \lceil \log_2 n \rceil - 1$ **do**
  for ▷ $j = 0, \dots, J_r - 1$ **do**
    for ▶ $k = 0, \dots, K-1$ **do**
      append $J + j$ to $L^{\mathbf{B}}$
  $K \leftarrow \lceil K/2 \rceil$ and $J \leftarrow J + J_r$
return $L^{\mathbf{B}}$ ▷ `(0,0,0,...)`

Algorithm 4 *Advance-first* target pulls $L^{\mathbf{A}}$

input number of arms: n, budget: T
initialize empty $L^{\mathbf{A}}$, $K \coloneqq n$, $J \coloneqq 0$
for $r = 0, \dots \lceil \log_2 n \rceil - 1$ **do**
  for ▶ $k = 0, \dots, K-1$ **do**
    for ▷ $j = 0, \dots, J_r - 1$ **do**
      append $J + j$ to $L^{\mathbf{A}}$
  $K \leftarrow \lceil K/2 \rceil$ and $J \leftarrow J + J_r$
return $L^{\mathbf{A}}$ ▷ `(0,1,2,...)`

the SH algorithm so that it can be easily extended to the batched setting (Sec. 3.1). Then, we consider *Breadth-first Sequential Halving* (BSH), one of the simplest batched extensions of SH, as an intermediate step (Sec. 3.2). Finally, we introduce *Advance-first Sequential Halving* (ASH) as a further extension (Sec. 3.3).

3.1 SH implementation with target pulls

Since BSH/ASH is a natural batched extension of SH, we first reconstruct the implementation of the SH algorithm as Algorithm 2 so that it can be easily extended to BSH/ASH. Note that, in this study, the operation $\mathrm{argmax}_{x \in \mathcal{X}}(\ell_x, m_x)$ selects the element $x \in \mathcal{X}$ that maximizes ℓ_x first. If multiple elements achieve this maximum, it then selects among these the one that maximizes m_x. At the t-th arm pull, SH selects the arm a_t that has the highest empirical reward $\bar{\mu}_a$ among the candidates $\mathcal{A}_t$:

$$a_t \coloneqq \mathrm{argmax}_{a \in \mathcal{A}_t} \bar{\mu}_a,$$

where $\mathcal{A}_t \coloneqq \{a \in [n] \mid N_a = L_t\}$ are the candidates at the t-th arm pull, N_a is the total number of pulls of arm a, and L_t is the number of *target pulls* at t, defined as either **breadth-first** manner

$$L_t^{\mathbf{B}} \coloneqq \underbrace{\sum_{r' < r(t)} J_{r'}}_{\text{pulls before } r(t)} + \underbrace{\left\lfloor \frac{t - \sum_{r' < r(t)} T_{r'}}{|\mathcal{S}_{r(t)}|} \right\rfloor}_{\text{pulls in } r(t)}, \tag{1}$$

or **advance-first** manner

$$L_t^{\mathbf{A}} \coloneqq \underbrace{\sum_{r' < r(t)} J_{r'}}_{\text{pulls before } r(t)} + \underbrace{\left(\left(t - \sum_{r' < r(i)} T_{r'} \right) \bmod J_{r(t)} \right)}_{\text{pulls in } r(t)}, \tag{2}$$

where $r(t)$ is the round of the t-th arm pull. This $L_t^{\mathbf{B}}/L_t^{\mathbf{A}}$ represents the cumulative number of pulls of the arm selected at the t-th pull before the t-th arm pull. We omitted the dependency on n and

Algorithm 5 ASH: Advance-first Sequential Halving

1: **input** number of arms: n, batch size: b, batch budget: B
2: **initialize** counter $t \coloneqq 0$, empirical mean $\bar{\mu}_a \coloneqq 0$, and arm pulls $N_a \coloneqq 0$ for all $a \in [n]$
3: **for** B times **do**
4:     initialize empty batch $\mathcal{B}$ and virtual arm pulls $M_a = 0$ for all $a \in [n]$
5:     **for** b times **do**
6:         let $\mathcal{A}_t$ be $\{a \in [n] \mid N_a + M_a = L_t^{\mathbf{A}}\}$ ▷ BSH uses $L_t^{\mathbf{B}}$ instead
7:         push $a_t \coloneqq \operatorname{argmax}_{a \in \mathcal{A}_t}(N_a, \bar{\mu}_a)$ to $\mathcal{B}$ ▷ BSH uses $\operatorname{argmax}_{a \in \mathcal{A}_t} \bar{\mu}_a$ instead
8:         update $t \leftarrow t + 1$ and $M_{a_t} \leftarrow M_{a_t} + 1$
9:     batch pull arms in $\mathcal{B}$
10:     update $\bar{\mu}_a$ and $N_a \leftarrow N_a + M_a$ for all $a \in \mathcal{B}$
11: **return** $\operatorname{argmax}_{a \in [n]}(N_a, \bar{\mu}_a)$

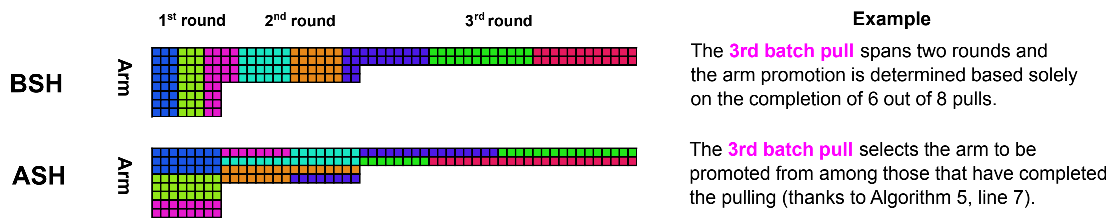

Figure 1: Pictorial representation of *breadth-first* SH (BSH; Sec. 3.2) and *advance-first* SH (ASH; Sec. 3.3) for an 8-armed bandit problem. Batch size b is 24 and batch budget B is 8. The same color indicates the same batch pull — For example, in the first batch pull (blue), BSH pulls each of the 8 arms 3 times, while ASH pulls 3 arms 8 times each. BSH selects arms so that the number of pulls of each active arm becomes as equal as possible, while ASH selects arms so that once an arm is selected, it is pulled until the budget for the arm in the round is exhausted. These pull sequences are characterized by the target pulls $L^{\mathbf{B}}$ and $L^{\mathbf{A}}$:

$L^{\mathbf{B}} = (0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,2,2,2,2,2,2,2,2,3,3,3,3,3,3,3,3,4,4,4,4,4,4,4,4,5,5,5,5,5,5,\ldots)$
$L^{\mathbf{A}} = (0,1,2,3,4,5,6,7,0,1,2,3,4,5,6,7,0,1,2,3,4,5,6,7,0,1,2,3,4,5,6,7,0,1,2,3,4,5,6,7,0,1,2,3,4,5,\ldots)$

T for simplicity. The definition of $L_t^{\mathbf{B}}/L_t^{\mathbf{A}}$ is somewhat complicated, and it may be straightforward to write down the algorithm that constructs $L^{\mathbf{B}} \coloneqq (L_0^{\mathbf{B}}, \ldots, L_T^{\mathbf{B}})$ and $L^{\mathbf{A}} \coloneqq (L_0^{\mathbf{A}}, \ldots, L_T^{\mathbf{A}})$ as shown in Algorithm 3 and Algorithm 4, respectively. Note that the choice between $L^{\mathbf{B}}$ and $L^{\mathbf{A}}$ is arbitrary and does not affect the behavior of SH — as long as the arm pull is sequential (not batched). Python code for this SH implementation is available in App. A. Note that using target pulls to implement SH is natural and not new. For example, Mctx[1] (Babuschkin et al., 2020) has a similar implementation.

3.2 BSH: Breadth-first Sequential Halving

Now, we extend SH to BSH, in which we select arms so that the number of pulls of each arm becomes as equal as possible using $L^{\mathbf{B}}$. Note that $L^{\mathbf{B}}$ uses $T = b \times B$ as the scheduled total budget. When pulling arms in a batch, we need to consider not only the number of pulls of the arms but also the number of scheduled pulls in the current batch. Therefore, we introduce *virtual arm pulls* M_a, the number of scheduled pulls of arm a in the current batch. For each batch pull, we sequentially select b arms with the highest empirical rewards from the candidates $\{a \in [n] \mid N_a + M_a = L_t^{\mathbf{B}}\}$ and pull them as a batch. The BSH algorithm is described in App. B. BSH is similar to a batched extension of SH introduced in Jun et al. (2016) in the sense that it selects arms so that the number of pulls of each arm becomes as equal as possible.

[1] `https://github.com/google-deepmind/mctx`

3.3 ASH: Advance-first Sequential Halving

We further extend SH to ASH in a manner similar to BSH. The ASH algorithm is described in Algorithm 5. Fig. 1 shows the pictorial representation of BSH and ASH. Python code for this ASH implementation is available in App. A. The differences between BSH and ASH are that:

1. ASH selects arms in *advance-first* manner using $L^{\mathbf{A}}$ instead of $L^{\mathbf{B}}$ (line 6), and
2. ASH considers not only the empirical rewards $\bar{\mu}_a$ but also the number of actual pulls N_a when selecting arms in a batch (line 7).

The second difference ensures that, when the batch spans two rounds, the arm to be promoted is selected from the arms that have completed pulling (e.g., see the 3rd batch pull in Fig. 1). Note that this second modification is not useful for BSH. Let $\pi_{\text{ASH}} : [0, 1]^{n\times T} \to [n]$ be the mapping induced by the ASH algorithm. In Sec. 4, we will show that ASH is algorithmically equivalent to SH with the same total budget $T = b \times B$ — π_{ASH} is identical to π_{SH}.

4 Algorithmic Equivalence of SH and ASH

This section presents a theoretical guarantee for the ASH algorithm.

Theorem 1 *Given a stochastic bandit problem with $n \geq 2$ arms, let $b \geq 2$ be the batch size and B be the batch budget satisfying $B \geq \max\{4, \frac{n}{b}\}\lceil \log_2 n\rceil$. Then, the ASH algorithm (Algorithm 5) is algorithmically equivalent to the SH algorithm (Algorithm 2) with the same total budget $T = b \times B$ — the mapping π_{ASH} is identical to π_{SH}.*

Proof sketch A key observation is that ASH and SH differ only when a batch pull spans two rounds, like the 3rd batch pull in Fig. 1. In this case, ASH may promote an incorrect arm to the next round that would not have been promoted in SH. We can prove that such *incorrect promotion* does not occur under the condition $B \geq \max\{4, \frac{n}{b}\}\lceil \log_2 n\rceil$. This is done by demonstrating that the inequality (3) holds for any $z < b$, the number of pulls for the current round r in the batch. Fig. 2 illustrates (3).

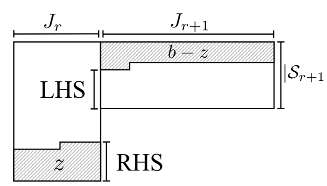

Figure 2: Inequality (3).

Proof. The condition $B \geq \max\{4, \frac{n}{b}\}\lceil \log_2 n\rceil$ is divided into two separate conditions:

$$B \geq \frac{n}{b}\lceil \log_2 n\rceil, \tag{C1}$$

and

$$B \geq 4\lceil \log_2 n\rceil. \tag{C2}$$

We focus on the scenario where a batch pull spans two rounds. In this case, let $z < b$ be the number of pulls that consume the budget for round r, and $b - z$ be the number of pulls that consume the budget for round $r+1$. The following proposition is demonstrated: $\forall n \geq 2, \forall b \geq 2, \forall r < \lceil \log_2 n\rceil - 1$, $\forall z < b$, if (C1) and (C2) hold, then

$$|\mathcal{S}_{r+1}| - \left\lceil \frac{b-z}{J_{r+1}} \right\rceil \geq \left\lceil \frac{z}{J_r} \right\rceil. \tag{3}$$

The left-hand side (LHS) of (3) represents the number of arms promoting to the subsequent round post-batch pull, whereas the right-hand side (RHS) quantifies the arms pending completion of their pulls at the batch pull juncture. This inequality, if satisfied, ensures that, even when a batch spans two rounds, arms supposed to advance to the next round in SH are not left behind in ASH, i.e., no incorrect promotion occurs. Considering the scenario where $z = b - 1$ suffices, as it represents the worst-case condition. Let $x \coloneqq |S_r| \geq 3$ for the given $r < \lceil \log_2 n \rceil - 1$. Two cases are considered. **Case 1:** when $n \leq 4b$. Given that $J_r = \left\lfloor \frac{b \times B}{x \lceil \log_2 n \rceil} \right\rfloor \geq \lfloor 4b/x \rfloor$ as derived from (C2), it is sufficient to show

$$\left\lceil \frac{x}{2} \right\rceil - 1 \geq \left\lceil \frac{b-1}{\lfloor 4b/x \rfloor} \right\rceil \tag{4}$$

in $x \in [3, 4b]$. This assertion is directly supported by Lemma 1. **Case 2:** when $4b < n$. Given that $J_r = \left\lfloor \frac{b \times B}{x \lceil \log_2 n \rceil} \right\rfloor \geq \lfloor n/x \rfloor$ as derived from (C1), it is sufficient to show $\left\lceil \frac{x}{2} \right\rceil - 1 \geq \left\lceil \frac{n/4-1}{\lfloor n/x \rfloor} \right\rceil$ in $x \in [3, n]$. This conclusion follows by the same reasoning applied in Case 1. $\square$

Lemma 1 *For any integer $b \geq 2$, the inequality $\left\lceil \frac{x}{2} \right\rceil - 1 \geq \left\lceil \frac{b-1}{\lfloor 4b/x \rfloor} \right\rceil$ holds for all integers $x \in [3, 4b]$.*

The proof of Lemma 1 is in App. C. Here, we provide the visualization of (4) in Fig. 3 to intuitively show that Lemma 1 holds. Each colored line represents the RHS for different $b \leq 32$. One can see that the LHS is always greater than the RHS for any $x \in [3, 4b]$.

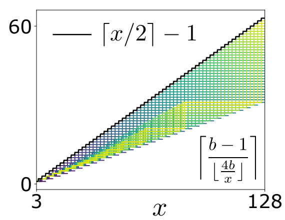

Figure 3: Lemma 1.

Remark 1 The condition (C1) is common to both SH and ASH — SH implicitly assumes $T \geq n \lceil \log_2 n \rceil$ as the minimum condition to execute. This is because we need to pull each arm at least once in the first round (i.e., $J_1 \geq 1$). With the same argument, the batch budget B must satisfy (C1). On the other hand, (C2) is specific to ASH and is required to ensure the equivalence. As we discuss in the Sec. 4.1, we argue that this additional (C2) is not practically problematic.

Remark 2 Note that the condition (C2) is tight; Theorem 1 does not hold even if $B \geq \alpha \lceil \log_2 n \rceil$ for any positive value $\alpha < 4$.

Proof. We aim to demonstrate the existence of a value x such that $\left\lceil \frac{x}{2} \right\rceil - 1 - \left\lceil \frac{b-1}{\lfloor \alpha b/x \rfloor} \right\rceil < 0$ when $n \leq \alpha b$. Consider the case when $x = 4$. In this scenario, the LHS of the inequality can be rewritten as $1 - \left\lceil \frac{b-1}{\lfloor \alpha b/4 \rfloor} \right\rceil \leq 1 - \frac{b-1}{\lfloor \alpha b/4 \rfloor} \leq 1 - \frac{4}{\alpha} \frac{b-1}{b} \to 1 - \frac{4}{\alpha}$ as $b \to \infty$. As $\alpha < 4$, it follows that LHS < 0 for sufficiently large values of b. $\square$

Remark 3 When b is sufficiently large, the minimum B that satisfies both (C1) and (C2) is $4 \lceil \log_2 n \rceil$. Theorem 1 implies that for arbitrarily large target budget T, ASH can achieve the same performance as SH by increasing the batch size b without increasing the batch budget B from $4 \lceil \log_2 n \rceil$ — ASH guarantees its scalability in batch computation.

Remark 4 Theorem 1 allows us to understand the properties of ASH based on existing theoretical research on SH, such as the simple regret bound (Zhao et al., 2023).

4.1 Discussion on the conditions

To show that SH and ASH are algorithmically equivalent, we used an additional condition (C2) of $\mathcal{O}(\log n)$. However, we argue that this condition is not practically problematic because the condition (C1), the minimum condition required to execute (unbatched) SH, is dominant ($\mathcal{O}(n \log n)$). This condition (C1) is dominant over (C2) as shown in Fig. 4. We can see that the condition (C2) only affects the algorithm when the batch size is sufficiently larger than the number of arms ($b \gg n$). This is a reasonable result, meaning that we cannot guarantee the equivalent behavior to SH with an extremely small batch budget, such as $B = 1$. On the other hand, if the user secures the minimum budget $B = 4 \lceil \log_2 n \rceil$ that depends only on the number of arms n and increases only logarithmically, regardless of the batch size b, they can increase the batch size arbitrarily and achieve the same result as when SH is executed sequentially with the same total budget, with high computational efficiency.

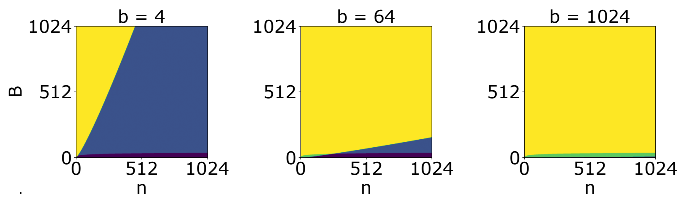

Figure 4: Visualization of conditions (C1) and (C2) for $n \leq 1024$, $B \leq 1024$, and $b \in \{4, 64, 1024\}$.

5 Empirical Validation

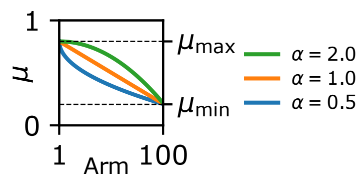

Figure 5: Polynomial(α)

We conducted experiments to empirically demonstrate that ASH maintains its performance for large batch size b, in comparison to its sequential counterpart SH. To evaluate this, we utilized a polynomial family parameterized by α as a representative batch problem instance, where the reward gap $\Delta_a := \mu_1 - \mu_a$ follows a polynomial distribution with parameter α: $\Delta_a \propto (a/n)^\alpha$ (Jamieson et al., 2013; Zhao et al., 2023). This choice is motivated by the observation that real-world applications exhibit polynomially distributed reward gaps, as mentioned in Zhao et al. (2023). In our study, we considered three different values of α (0.5, 1.0, and 2.0) to capture various reward distributions (see Fig. 5). Additionally, we characterized each bandit problem instance by specifying the minimum and maximum rewards, denoted as $\mu_{\min}$ and $\mu_{\max}$ respectively. Hence, we denote a bandit problem instance as $\mathcal{T}(n, \alpha, \mu_{\min}, \mu_{\max})$.

We also implemented a simple batched extension of SH introduced by Jun et al. (2016) as a baseline for comparison. We refer to this algorithm as Jun+16. The implementation of Jun+16 is described in App. D. Jun et al. (2016) did not provide a theoretical guarantee for Jun+16, but it has shown performance comparable to or better than their proposed algorithm in their experiments.

5.1 Large batch budget scenario: $B \geq 4\lceil \log_2 n \rceil$

First, we empirically confirm that, as we claimed in Sec. 4, ASH is indeed equivalent to SH under the condition (C2). We generated 10K instances of bandit problems and applied ASH and SH to each instance with 100 different seeds. We randomly sampled n from $\{2, \ldots, 1024\}$, α from $\{0.5, 1.0, 2.0\}$, and $\mu_{\min}$ and $\mu_{\max}$ from $\{0.1, 0.2, \ldots, 0.9\}$. For each instance $\mathcal{T}(n, \alpha, \mu_{\min}, \mu_{\max})$, we randomly sampled the batch budget $B \leq 10\lceil \log_2 n \rceil$ and the batch size $b \leq 5n$ so that the condition (C1) and (C2) are satisfied. *As a result, we confirmed that the selected arms of ASH and SH are identical in all 10K instances and 100 seeds for each instance.* We also conducted the same experiment for BSH and Jun+16. We plotted the simple regret of BSH, ASH, and Jun+16 against SH in Fig. 6. There are 10K instances, and each point represents the average simple regret of 100 seeds for each instance. To compare the performance, we fitted a linear regression model to the simple regret of BSH, ASH, and Jun+16 against SH as $y = \beta x$, where y is the simple regret of BSH, ASH, or Jun+16, x is the simple regret of SH. The slope β is estimated by the least squares method. The estimated slope β is 1.008 for BSH, 1.000 for ASH, and 0.971 for Jun+16, which indicates that the simple regret of ASH, BSH, and Jun+16 is comparable to SH on average.

5.2 Small batch budget scenario: $B < 4\lceil \log_2 n \rceil$

Next, we examined the performances of BSH, ASH, and Jun+16 against SH when the additional condition (C2) is not satisfied, i.e., when the batch budget is extremely small $B < 4\lceil \log_2 n \rceil$ and thus Theorem 1 does not hold. We conducted the same experiment as in Sec. 5.1 except the batch budget $B < 4\lceil \log_2 n \rceil$. We sampled B so that B is larger than the number of rounds. The results are

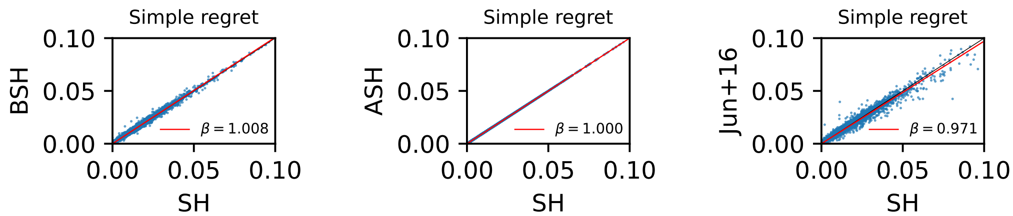

Figure 6: Single regret comparison of BSH, ASH, and Jun+16 against SH when $B \geq 4\lceil \log_2 n \rceil$.

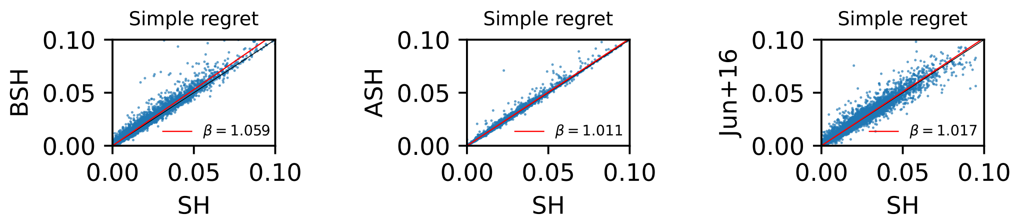

Figure 7: Single regret comparison of BSH, ASH, and Jun+16 against SH when $B < 4\lceil \log_2 n \rceil$.

shown in Fig. 7. The slope β is estimated as 1.059 for BSH, 1.011 for ASH, and 1.017 for Jun+16. All the estimated slopes are worse than when $B \geq 4\lceil \log_2 n \rceil$. However, the estimated slopes are still close to 1, which indicates that while we do not have a theoretical guarantee, the performance of BSH, ASH, and Jun+16 is comparable to SH on average.

6 Related Work

Sequential Halving. Among the algorithms for the pure exploration problem in multi-armed bandits (Audibert et al., 2010), Sequential Halving (SH; Karnin et al. (2013)) is one of the most popular algorithms. The theoretical properties of SH have been well studied (Karnin et al., 2013; Zhao et al., 2023). Due to its simplicity, SH has been widely used for these (but is not limited to) applications: In the context of *tree-search* algorithms, as the root node selection of Monte Carlo tree search can be regarded as a pure exploration problem (Tolpin & Shimony, 2012), Danihelka et al. (2022) incorporated SH into the root node selection and significantly reduced the number of simulations to improve the performance during AlphaZero/MuZero training. From the min-max search perspective, some studies recursively applied SH to the internal nodes of the search tree (Cazenave, 2014; Pepels et al., 2014). SH is also used for *hyperparameter optimization*; Jamieson & Talwalkar (2016) formalized the hyperparameter optimization problem in machine learning as a *non-stochastic* multi-armed bandit problem, where the reward signal is not from stochastic stationary distributions but from deterministic function changing over training steps. Li et al. (2018; 2020) applied SH to hyperparameter optimization in asynchronous parallel settings, which is similar to our batch setting. Their asynchronous approach may have *incorrect promotions* to the next rounds but is more efficient than the synchronous approach. Aziz et al. (2022) applied SH to *recommendation systems*, which identify appealing podcasts for users.

Batched bandit algorithms. Batched bandit algorithms have been studied in various contexts (Perchet et al., 2016; Gao et al., 2019; Esfandiari et al., 2021; Jin et al., 2021a;b; Kalkanli & Ozgur, 2021; Karbasi et al., 2021; Provodin et al., 2022). Among the batched bandit studies for the pure exploration problem (Agarwal et al., 2017; Grover et al., 2018; Jun et al., 2016), Jun et al. (2016) is the most relevant to our work as they also consider the *fixed-size batch pulls* setting. To the best of our knowledge, the first batched SH variant with a fixed batch size b was introduced by Jun et al. (2016) as a baseline algorithm in their study (Jun+16). It is similar to BSH and it pulls arms so that the number of pulls of the arms is as equal as possible (breadth-first manner). They reported that Jun+16 experimentally performs comparably to or better than their proposed method but did not provide a theoretical guarantee for Jun+16. Our ASH is different from their batch variant in that ASH pulls arms in an advance-first manner instead of a breadth-first manner.

7 Limitation and Future Work

Our batched variants of SH assume that the reward distributions of the arms are from i.i.d. distributions. This property is essential to allow batch pulls. One limitation is that it may be difficult to apply our algorithms to bandit problems where the reward distribution is non-stationary. For example, Jamieson & Talwalkar (2016) applied SH to hyperparameter tuning, where rewards are time-series losses during model training. We cannot apply our batched variants to this problem because we cannot observe "future losses" in a batch.

Our batched variants of SH are suitable for tasks where arms can be evaluated efficiently in batches rather than sequentially. For instance, when the evaluation of arms depends on the output of neural networks, the process can be efficiently conducted in batches using accelerators like GPUs. An example of this scenario is provided by Danihelka et al. (2022), where value networks are used in Monte Carlo tree search. Applying our batched variants to such algorithms is a possible future direction. Additionally, combining them with reinforcement learning environments that run on GPU/TPU accelerators (Freeman et al., 2021; Lange, 2022; Koyamada et al., 2023; Gulino et al., 2023; Nikulin et al., 2023; Bonnet et al., 2024; Rutherford et al., 2024; Matthews et al., 2024) for efficient batch evaluation is also promising.

8 Conclusion

In this paper, we proposed ASH as a simple and natural extension of the SH algorithm. We theoretically showed that ASH is algorithmically equivalent to SH as long as the batch budget is not excessively small. This allows ASH to inherit the well-studied theoretical properties of SH, including the simple regret bound. Our experimental results confirmed this claim and demonstrated that ASH and other batched variants of SH, like Jun+16, perform comparably to SH in terms of simple regret. These findings suggest that we can utilize simple batched variants of SH for efficient evaluation of arms with large batch sizes while avoiding performance degradation compared to the sequential execution of SH. By providing a practical solution for efficient arm evaluation, our study opens up new possibilities for applications that require large budgets. Overall, our work highlights the batch robust nature of SH and its potential for large-scale bandit problems.

Broader Impact Statement

The findings in this work on the bandit problem are focused on theoretical results and do not involve direct human or ethical implications. Therefore, concerns related to broader ethical, humanitarian, and societal issues are not applicable to this research. However, if our approach is applied to large-scale bandit problems, especially when batch evaluation involves large neural networks, there could be an indirect impact on energy consumption due to the computational resources required.

Acknowledgments

This paper is based on results obtained from a project, JPNP20006, subsidized by the New Energy and Industrial Technology Development Organization (NEDO), and partly supported by KAKENHI (No. 22H04998 and 23H04676) from Japan Society for the Promotion of Science (JSPS). We sincerely thank the reviewers for their invaluable feedback and constructive comments, which have significantly enhanced the quality of this research. We would also like to express our gratitude to the developers of the software libraries utilized in this research, including NumPy (Harris et al., 2020), SciPy (Virtanen et al., 2020), Matplotlib (Hunter, 2007), JAX (Bradbury et al., 2018), and Mctx (Babuschkin et al., 2020).

References

Arpit Agarwal, Shivani Agarwal, Sepehr Assadi, and Sanjeev Khanna. Learning with Limited Rounds of Adaptivity: Coin Tossing, Multi-Armed Bandits, and Ranking from Pairwise Comparisons. In *COLT*, 2017.

Jean-Yves Audibert, Sébastien Bubeck, and Rémi Munos. Best Arm Identification in Multi-armed Bandits. In *COLT*, 2010.

Maryam Aziz, Jesse Anderton, Kevin Jamieson, Alice Wang, Hugues Bouchard, and Javed Aslam. Identifying new podcasts with high general appeal using a pure exploration infinitely-armed bandit strategy. In *RecSys*, 2022.

Igor Babuschkin, Kate Baumli, Alison Bell, Surya Bhupatiraju, Jake Bruce, Peter Buchlovsky, David Budden, Trevor Cai, Aidan Clark, Ivo Danihelka, Claudio Fantacci, Jonathan Godwin, Chris Jones, Ross Hemsley, Tom Hennigan, Matteo Hessel, Shaobo Hou, Steven Kapturowski, Thomas Keck, Iurii Kemaev, Michael King, Markus Kunesch, Lena Martens, Hamza Merzic, Vladimir Mikulik, Tamara Norman, John Quan, George Papamakarios, Roman Ring, Francisco Ruiz, Alvaro Sanchez, Rosalia Schneider, Eren Sezener, Stephen Spencer, Srivatsan Srinivasan, Luyu Wang, Wojciech Stokowiec, and Fabio Viola. The DeepMind JAX Ecosystem. https://github.com/google-deepmind, 2020.

Clément Bonnet, Daniel Luo, Donal Byrne, Shikha Surana, Vincent Coyette, Paul Duckworth, Laurence I. Midgley, Tristan Kalloniatis, Sasha Abramowitz, Cemlyn N. Waters, Andries P. Smit, Nathan Grinsztajn, Ulrich A. Mbou Sob, Omayma Mahjoub, Elshadai Tegegn, Mohamed A. Mimouni, Raphael Boige, Ruan de Kock, Daniel Furelos-Blanco, Victor Le, Arnu Pretorius, and Alexandre Laterre. Jumanji: a Diverse Suite of Scalable Reinforcement Learning Environments in JAX. *ICLR*, 2024.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs. https://github.com/google/jax, 2018.

Sébastien Bubeck, Rémi Munos, and Gilles Stoltz. Pure Exploration in Multi-armed Bandits Problems. In *ALT*, 2009.

Tristan Cazenave. Sequential Halving Applied to Trees. *IEEE T-CIAIG*, 7(1):102–105, 2014.

Ivo Danihelka, Arthur Guez, Julian Schrittwieser, and David Silver. Policy improvement by planning with Gumbel. In *ICLR*, 2022.

Hossein Esfandiari, Amin Karbasi, Abbas Mehrabian, and Vahab Mirrokni. Regret Bounds for Batched Bandits. In *AAAI*, 2021.

Daniel Freeman, Erik Frey, Anton Raichuk, Sertan Girgin, Igor Mordatch, and Olivier Bachem. Brax - A Differentiable Physics Engine for Large Scale Rigid Body Simulation. In *NeurIPS Track on Datasets and Benchmarks*, 2021.

Zijun Gao, Yanjun Han, Zhimei Ren, and Zhengqing Zhou. Batched Multi-armed Bandits Problem. In *NeurIPS*, 2019.

Aditya Grover, Todor Markov, Peter Attia, Norman Jin, Nicolas Perkins, Bryan Cheong, Michael Chen, Zi Yang, Stephen Harris, William Chueh, and Stefano Ermon. Best arm identification in multi-armed bandits with delayed feedback. In *AISTATS*, 2018.

Cole Gulino, Justin Fu, Wenjie Luo, George Tucker, Eli Bronstein, Yiren Lu, Jean Harb, Xinlei Pan, Yan Wang, Xiangyu Chen, John Co-Reyes, Rishabh Agarwal, Rebecca Roelofs, Yao Lu, Nico Montali, Paul Mougin, Zoey Yang, Brandyn White, Aleksandra Faust, Rowan McAllister, Dragomir Anguelov, and Benjamin Sapp. Waymax: An Accelerated, Data-Driven Simulator for Large-Scale Autonomous Driving Research. In *NeurIPS*, 2023.

Charles R Harris, K Jarrod Millman, Stéfan J Van Der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020.

John D. Hunter. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, 9 (3):90–95, 2007.

Kevin Jamieson and Ameet Talwalkar. Non-stochastic Best Arm Identification and Hyperparameter Optimization. In *AISTATS*, 2016.

Kevin Jamieson, Matthew Malloy, Robert Nowak, and Sebastien Bubeck. On Finding the Largest Mean Among Many. *arXiv:1306.3917*, 2013.

Tianyuan Jin, Jing Tang, Pan Xu, Keke Huang, Xiaokui Xiao, and Quanquan Gu. Almost Optimal Anytime Algorithm for Batched Multi-Armed Bandits. In *ICML*, 2021a.

Tianyuan Jin, Pan Xu, Xiaokui Xiao, and Quanquan Gu. Double Explore-then-Commit: Asymptotic Optimality and Beyond. In *COLT*, 2021b.

Kwang-Sung Jun, Kevin Jamieson, Robert Nowak, and Xiaojin Zhu. Top Arm Identification in Multi-Armed Bandits with Batch Arm Pulls. In *AISTATS*, 2016.

Cem Kalkanli and Ayfer Ozgur. Batched Thompson Sampling. In *NeurIPS*, 2021.

Amin Karbasi, Vahab Mirrokni, and Mohammad Shadravan. Parallelizing Thompson Sampling. In *NeurIPS*, 2021.

Zohar Karnin, Tomer Koren, and Oren Somekh. Almost Optimal Exploration in Multi-Armed Bandits. In *ICML*, 2013.

Sotetsu Koyamada, Shinri Okano, Soichiro Nishimori, Yu Murata, Keigo Habara, Haruka Kita, and Shin Ishii. Pgx: Hardware-Accelerated Parallel Game Simulators for Reinforcement Learning. In *NeurIPS*, 2023.

Robert Tjarko Lange. gymnax: A JAX-based Reinforcement Learning Environment Library. http://github.com/RobertTLange/gymnax, 2022.

Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Jonathan Ben-tzur, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. A System for Massively Parallel Hyperparameter Tuning. In *MLSys*, 2020.

Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. *JMLR*, 18(185):1–52, 2018.

Michael Matthews, Michael Beukman, Benjamin Ellis, Mikayel Samvelyan, Matthew Jackson, Samuel Coward, and Jakob Foerster. Craftax: A Lightning-Fast Benchmark for Open-Ended Reinforcement Learning. *arXiv:2402.16801*, 2024.

Alexander Nikulin, Vladislav Kurenkov, Ilya Zisman, Viacheslav Sinii, Artem Agarkov, and Sergey Kolesnikov. XLand-MiniGrid: Scalable Meta-Reinforcement Learning Environments in JAX. In *NeurIPS 2023 Workshop*, 2023.

Tom Pepels, Tristan Cazenave, Mark HM Winands, and Marc Lanctot. Minimizing Simple and Cumulative Regret in Monte-Carlo Tree Search. In *CGW*, 2014.

Vianney Perchet, Philippe Rigollet, Sylvain Chassang, and Erik Snowberg. Batched bandit problems. *Ann. Stat.*, 44(2):660 – 681, 2016.

D. Provodin, P. Gajane, M. Pechenizkiy, and M. Kaptein. The Impact of Batch Learning in Stochastic Linear Bandits. In *ICDM*, 2022.

Alexander Rutherford, Benjamin Ellis, Matteo Gallici, Jonathan Cook, Andrei Lupu, Garðar Ingvarsson, Timon Willi, Akbir Khan, Christian Schroeder de Witt, Alexandra Souly, et al. Jax-MARL: Multi-Agent RL Environments and Algorithms in JAX. In *AAMAS*, 2024.

Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering Atari, Go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359, 2017.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144, 2018.

David Tolpin and Solomon Shimony. MCTS Based on Simple Regret. In *AAAI*, 2012.

Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature methods*, 17(3):261–272, 2020.

Yao Zhao, Connor Stephens, Csaba Szepesvari, and Kwang-Sung Jun. Revisiting Simple Regret: Fast Rates for Returning a Good Arm. In *ICML*, 2023.

A Python code

For the sake of reproducibility and a better understanding, we provide Python code for the Sequential Halving (SH) algorithm using advance-first target pulls and the Advance-first Sequential Halving (ASH) algorithm in Fig. 8.

```python
from math import log2, ceil, floor
import numpy as np

def sh(bandit: BanditProblem, n: int, T: int) -> int:
    L = _get_target_pulls(n, T)                   # L: target pulls
    N = np.append(np.zeros(n, dtype=int), -1e9)   # N: pull counts
    R = np.append(np.zeros(n, dtype=float), 0.)   # R: avg rewards
    for t in range(T):
        a = np.argmax(np.where(N == L[t], R, -np.inf))
        r = bandit.pull(a)
        R[a] = (R[a] * N[a] + r) / (N[a] + 1)
        N[a] += 1
    return int(np.argmax(np.where(N >= max(N), R, -np.inf)))

def ash(bandit: BanditProblem, n: int, B: int, b: int = 1) -> int:
    L = _get_target_pulls(n, b * B)               # L: target pulls
    N = np.append(np.zeros(n, dtype=int), -1e9)   # N: pull counts
    R = np.append(np.zeros(n, dtype=float), 0.)   # R: avg rewards
    for i in range(B):
        batch = []
        M = np.zeros_like(N)                      # M: virtual pull counts
        for j in range(b):
            t = i * b + j
            N_max = np.max(np.where(N + M == L[t], N, -np.inf))
            a = np.argmax(np.where((N + M == L[t]) & (N == N_max), R, -np.inf))
            batch.append(a)
            M[a] += 1
        rewards = bandit.batch_pull(batch)
        for a, r in zip(batch, rewards):
            R[a] = (R[a] * N[a] + r) / (N[a] + 1)
            N[a] += 1
    return int(np.argmax(np.where(N >= max(N), R, -np.inf)))

def _get_target_pulls(n: int, T: int) -> list[int]:  # advance-first
    target_pulls = []
    num_rounds = ceil(log2(n))
    num_active_arms = n
    cum_pulls = 0
    for r in range(num_rounds):
        J = floor(T / (num_active_arms * num_rounds))
        if r == num_rounds - 1:
            remaining_pulls = T - len(target_pulls)
            J = remaining_pulls // 2
        for _ in range(num_active_arms):
            for i in range(J):
                target_pulls.append(cum_pulls + i)
        cum_pulls += J
        num_active_arms = ceil(num_active_arms / 2)  # halving
    return target_pulls + [int(-1e9)] * (T - len(target_pulls))
```

Figure 8: Python implementation of the SH algorithm using advance-first target pulls (Algorithm 2) and the ASH algorithm (Algorithm 5).

B BSH algorithm

Algorithm 6 shows the detailed BSH algorithm (see Sec. 3.2).

Algorithm 6 BSH: Breadth-first Sequential Halving

input number of arms: n, batch size: b, batch budget: B
initialize counter $t \coloneqq 0$, empirical mean $\bar{\mu}_a \coloneqq 0$, and arm pulls $N_a \coloneqq 0$ for all $a \in [n]$
for B times **do**
  initialize empty batch $\mathcal{B}$ and virtual arm pulls $M_a = 0$ for all $a \in [n]$
  for b times **do**
    let $\mathcal{A}_t$ be $\{a \in [n] \mid N_a + M_a = L_t^{\mathbf{B}}\}$
    push $a_t \coloneqq \mathrm{argmax}_{a \in \mathcal{A}_t} \bar{\mu}_a$ to $\mathcal{B}$
    update $t \leftarrow t + 1$ and $M_{a_t} \leftarrow M_{a_t} + 1$
  batch pull arms in $\mathcal{B}$
  update $\bar{\mu}_a$ and $N_a \leftarrow N_a + M_a$ for all $a \in \mathcal{B}$
return $\mathrm{argmax}_{a \in [n]} (N_a, \bar{\mu}_a)$

C Proof of Lemma 1

Lemma 1 *For any integer $b \geq 2$, the inequality*

$$\left\lceil \frac{x}{2} \right\rceil - 1 \geq \left\lceil \frac{b-1}{\lfloor 4b/x \rfloor} \right\rceil \tag{5}$$

holds for all integers $x \in [3, 4b]$.

Proof. This proof demonstrates that for any integer $b \geq 2$ and $x \in [3, 4b]$, the inequality (5) is satisfied. Given $z \geq c \implies z \geq \lceil c \rceil$ for any integer z and real number c, it suffices to demonstrate that

$$\left\lceil \frac{x}{2} \right\rceil - 1 \geq \frac{b-1}{\lfloor 4b/x \rfloor} \iff \left\lceil \frac{x}{2} \right\rceil - 1 - \frac{b-1}{\lfloor 4b/x \rfloor} \geq 0.$$

Given that $\left\lfloor \frac{4b}{x} \right\rfloor > 0$, it follows that

$$\left(\left\lceil \frac{x}{2} \right\rceil - 1 \right) \left\lfloor \frac{4b}{x} \right\rfloor - (b-1) \geq 0, \tag{6}$$

for any integer $b \geq 2$ and $x \in [3, 4b]$. Two cases are considered:

Case 1: x is even. Suppose $x = 2y$, with $y \in [2, 2b]$. We aim to show that

$$(y-1) \left\lfloor \frac{2b}{y} \right\rfloor - (b-1) \geq 0. \tag{7}$$

Two sub-cases are considered:

1. For $y \in [b+1, 2b]$, as $\left\lfloor \frac{2b}{y} \right\rfloor = 1$, LHS $= (y-1) - (b-1) \geq 0$.
2. For $y \in [2, b]$, as $\lfloor c \rfloor > c - 1$ for any real number c, we have LHS $> (y-1) \left(\frac{2b}{y} - 1 \right) - (b-1) = -\frac{(y-2)(y-b)}{y}$. As $y > 0$ and $-(y-2)(y-b) \geq 0$ in $y \in [2, b]$, we have LHS ≥ 0.

Consequently, it has been established that for even values of x, the inequality (7) is upheld.

Case 2: x is odd. Suppose $x = 2y + 1$, with $y \in [1, 2b - 1]$. We aim to show that

$$y \left\lfloor \frac{4b}{2y+1} \right\rfloor - (b-1) \geq 0. \tag{8}$$

Two sub-cases are considered:

1. For $y \in [b, 2b-1]$, as $\left\lfloor \frac{4b}{2y+1} \right\rfloor = 1$, LHS $= y - (b-1) \geq 0$.
2. For $y \in [1, b-1]$, as $\lfloor c \rfloor > c-1$ for any real number c, we have LHS $> y\left(\frac{4b}{2y+1} - 1\right) - (b-1) = \frac{2by-b-2y^2+y+1}{2y+1} = \frac{-2y(y-(b+\frac{1}{2}))-(b-1)}{2y+1} \geq 0$. As $2y+1 > 0$ and $-2y(y-(b+\frac{1}{2})) - (b-1) \geq 0$ in $y \in [1, b-1]$, we have LHS ≥ 0.

Similarly, it has been demonstrated that for odd values of x, the inequality (8) is upheld.

Therefore, through the analysis of these two cases, it is proven that for any integer $b \geq 2$ and $x \in [3, 4b]$, the inequality (6) is satisfied, thereby confirming the validity of (5). □

D Batch Sequential Halving introduced in Jun et al. (2016)

Algorithm 7 shows the detailed batched version of the Sequential Halving algorithm introduced in Jun et al. (2016).

Algorithm 7 Batched Sequential Halving introduced in Jun et al. (2016)

```
input number of arms: n, batch budget: B, batch size: b
initialize best arm candidates S_0 := [n]
for round r = 0, ..., ⌈log_2 n⌉ − 1 do
    for ⌊B/⌈log_2 n⌉⌋ times do
        select batch actions 𝓑 so that the number of pulls of each arm in S_r is as equal as possible
        pull arms 𝓑 in the batch
    S_{r+1} ← top-⌈|S_r|/2⌉ arms in S_r w.r.t. the empirical rewards
return the only arm in S_{⌈log_2 n⌉}
```