Computing Whittle (and Gittins) Index in Subcubic Time

Nicolas Gast, Bruno Gaujal, Kimang Khun

▶ **To cite this version:**

Nicolas Gast, Bruno Gaujal, Kimang Khun. Computing Whittle (and Gittins) Index in Subcubic Time. 2022. hal-03602458v1

HAL Id: hal-03602458

https://inria.hal.science/hal-03602458v1

Preprint submitted on 9 Mar 2022 (v1), last revised 20 Jun 2023 (v5)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Computing Whittle (and Gittins) Index in Subcubic Time

NICOLAS GAST, BRUNO GAUJAL, and KIMANG KHUN, Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP*, LIG, 38000 Grenoble, France

*Institute of Engineering Univ. Grenoble Alpes

Whittle index is a generalization of Gittins index that provides very efficient allocation rules for restless multi-armed bandits. In this paper, we develop an algorithm to test the indexability and compute the Whittle indices of any finite-state Markovian bandit problem. This algorithm works in the discounted and non-discounted cases. As a byproduct, it can also be used to compute Gittins index. Our algorithm builds on three tools: (1) a careful characterization of Whittle index that allows one to compute recursively the kth smallest index from the $(k-1)$th smallest, and to test indexability, (2) the use of Sherman-Morrison formula to make this recursive computation efficient, and (3) a sporadic use of fast matrix inversion and multiplication to obtain a subcubic complexity. We show that an efficient use of the Sherman-Morrison formula leads to an algorithm that computes Whittle index in $(2/3)n^3 + o(n^3)$ arithmetic operations, where n is the number of states of the arm. The careful use of fast matrix multiplication leads to the first subcubic algorithm to compute Whittle (or Gittins) index. By using the current fastest matrix multiplications, our algorithm runs in $O(n^{2.5286})$. We also conduct a series of experiments that demonstrate that our algorithm is very efficient in practice and can compute indices of Markov chains with several thousands of states in a few seconds.

1 INTRODUCTION

Markovian bandits form a subclass of multi-armed bandit problems in which each arm has an internal state that evolves over time in a Markovian manner, as a function of the decision maker's actions. In such a problem, at each time step, the decision maker observes the state of all arms and chooses which one to activate. If each arm has up to n states, then an M-arm problem can be represented by a Markov decision process (MDP) with n^M states. When the state of an arm evolves only when this arm is chosen, one falls into the category of *rested* Markovian bandits for which an optimal policy (in the discounted case) was found by Gittins [23]. When the state of an arm can also evolve when the arm is not chosen, the problem is called a *restless bandit* problem, and computing an optimal policy is a difficult problem [40].

In his seminal paper [47], Whittle proposed a very efficient heuristic: For each arm, an index function maps each state of this arm to a real number, the Whittle index policy then consists in activating the arms having the highest index first. This heuristic generalizes Gittins index to restless bandits. Contrary to the rested case, the Whittle index policy is in general not optimal. Yet, this policy has been proven to be very efficient over the years: up to a condition called *indexability*, Whittle index has been shown to be (in the undiscounted case) asymptotically optimal as the number of arms grows to infinity [33, 46]. Moreover, the heuristic performs extremely well in practice [9, 24, 26]. Restless bandits and Whittle index have been applied to many scheduling and resource allocation problems such as wireless communication [3, 32], web crawling [12, 37], congestion control [11, 13], queueing systems [1, 2, 10, 15, 25, 31, 42], and clinical trials [45].

The good performance of Whittle index for indexable bandits raises two important questions:

Is there an efficient algorithm to test indexability and compute Whittle index?
Is Whittle index harder to compute than Gittins index?

Related work. The computation of Gittins index has received a lot of attention in the past, see for instance [17, 30, 35, 43] and the recent survey [16]. For a n-state Markovian bandit, the

Authors' address: Nicolas Gast, nicolas.gast@inria.fr; Bruno Gaujal, bruno.gaujal@inria.fr; Kimang Khun, kimang.khun@inria.fr, Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP*, LIG, 38000 Grenoble, France
*Institute of Engineering Univ. Grenoble Alpes.

algorithm having the smallest complexity is the *fast-pivoting* implementation of [35] that performs $(2/3)n^3 + O(n^2)$ arithmetic operations. Note that the same author claims in page 4 of [38] that it is unlikely that this complexity can be improved. As we see later, we do improve upon this complexity.

The situation of Whittle index is more contrasted. To the best of our knowledge, there is no general purpose algorithm to test indexability, and most papers studying Whittle index either assume that the studied model is indexable or focus on specific classes of restless bandits for which the structure can be used to show indexability, see *e.g.* [2, 4, 7, 15]. Assuming indexability, the computation of Whittle index has been considered by a few papers.

For restless bandits with discount, $\beta \in (0, 1)$, the most efficient numerical algorithm to compute Whittle index was recently presented in [38]. This algorithm, called *fast-pivoting* algorithm, performs $(2/3)n^3 + O(n^2)$ arithmetic operations[1] if the initialization phase is excluded from the count. This is done by using the parametric simplex method and exploiting the special structure of this linear system to reduce the complexity of simplex pivoting steps. This fast-pivoting algorithm is an efficient implementation of adaptive-greedy algorithm [36] which outputs Whittle index if and only if the restless bandit satisfies a technical condition called partial conservation law (PCL), which is more restrictive than just being indexable. So, it is not applicable for all indexable restless bandits. Based on a geometric interpretation of Whittle index, the authors in [5] propose a refinement of the adaptive-greedy algorithm of [36] to compute Whittle indices of all indexable restless bandits. The refined algorithm achieves a $O(n^3)$ complexity by using the Sherman-Morrison formula. The authors also propose several checkable conditions to test indexability. These conditions are not necessary for indexability, which means that if an arm does not verify the conditions, we cannot conclude that the arm is non-indexable and an algorithm to check indexability is still needed. Also, no detailed description is given for adapting these conditions and their algorithm to restless bandit without discount. A thorough comparison between our algorithm and [5, 38] is given in Appendix D. While computing the Whittle indices of a known arm's model is still a challenge, there is interesting work in trying to learn Whittle index when only the arm's simulator is given and the arm's model is unknown. For instance, [12, 20] use Q-learning algorithm to estimate Whittle index as time evolves in finite-state restless bandit problems. Moreover, the work of [34] uses deep reinforcement learning framework to estimate the Whittle indices of the arms with large state space or convoluted transition kernel. With "strongly indexable" property introduced by the authors, deadline scheduling, recovering bandits, and wireless scheduling problems are considered and their method numerically outperforms or matches state-of-the-art control policies in each of the three problems.

For non-discounted restless bandit, the author of [38] only provides a brief description of how the fast-pivoting algorithm given in the discounted case can be adapted although no explicit algorithm is given in that paper. For continuous-time n-state restless bandits, the work of [14] proposes an algorithm to check indexability and compute Whittle index with a complexity exponential in the number of states n of each arm. According to Remark 4.1 of that paper, this complexity can be reduced to $O(n^5)$ if the restless bandit is known to be indexable and threshold-based policies are optimal. It is stated that their approach is not applicable for discounted restless bandits.

Contributions. In this paper, we investigate Whittle index computation in restless bandit problems and present two main contributions. Our first contribution is to propose a unified algorithm that computes the Whittle indices of Markovian bandits for both discounted and non-discounted restless bandits. Our algorithm, which can be viewed as a refinement of the algorithm in [5], detects whether the input problem is indexable and computes Whittle index if the problem is. As a byproduct, our algorithm can compute Gittins index in rested bandits which are a subclass of restless bandits. This

[1] multiplications and additions of real numbers, regardless of their values

algorithm computes the indices in increasing order, and relies on an efficient the use of Sherman-Morrison formula to compute Whittle index in $(2/3)n^3+O(n^2)$ plus the time to solve a linear system of order n, which is subcubic [44]. This algorithm can detect on the fly if a computed index violates the indexability condition, which adds an extra $n^3/3+O(n^2)$ arithmetic operations. This later test is optional and therefore the complexity of the algorithm can be $n^3+o(n^3)$ or $(2/3)n^3+o(n^3)$ depending on whether testing the indexability is needed or not.

As a second contribution, we show how to reduce the complexity of the above algorithm to obtain the first subcubic algorithm to compute Whittle[2] index. This improvement is made possible by the fact that a linear system can be solved in subcubic time. By carefully reordering the computations, we show that it is possible to reduce the use of the Sherman-Morrison formula at the price of solving more linear systems. The subcubic complexity comes by striking a good balance between having too many or two few linear systems to solve. By using the matrix multiplication method having the current smallest complexity, our algorithm can test indexability and compute Whittle index in $O(n^{2.5286})$.

Finally, we present a detailed numerical study on the performance of our algorithm and compare it with some raw data taken from [38]. The reported results show that our algorithm is very efficient in computing Whittle index and testing indexability. Moreover, our simulations indicate that the subcubic version of our algorithm does not just have a theoretically small complexity but is also faster in practice than our original $(2/3)n^3$ algorithm. Testing the indexability and computing indices takes less than one second for $n=1000$ states and less than 10 minutes for $n=15000$ states.

Road map. The paper is organized as follows. We introduce the problem and recall the definition of Whittle index in Section 2. In Section 3, we characterize Whittle index and show how to use the Sherman-Morrison formula to compute it efficiently. We then show how to reduce the complexity of the algorithm by using fast matrix multiplication algorithms in Section 5. We show how to adapt this approach to the discounted case in Section 7. We compare the different variants of our algorithm in Section 6. Finally, we conclude in Section 8.

2 RESTLESS MARKOVIAN BANDIT AND INDEXABILITY

2.1 Restless Bandit Process and Restless Multi-armed Bandit

A discrete-time restless bandit process (RB) is a Markov decision process (MDP) with discrete state space $[n]:=\{1\dots n\}$ and binary action space $\{0,1\}$, where 0 denotes the action "rest" and 1 denotes the action "activate". The time is discrete and the evolution is Markovian: If the MDP is in state i and action a is chosen, the decision maker earns an instantaneous reward r_i^a and the process transitions to a new state j with probability P_{ij}^a. We denote this MDP by the couple $(\boldsymbol{r},\boldsymbol{P})$. A policy π for a RB is a subset of $[n]$ such that the policy chooses to activate the process in state i if $i\in\pi$. By abuse of notation, we note $\pi_i=1$ if $i\in\pi$ and $\pi_i=0$ if $i\notin\pi$.

A restless multi-armed bandit problem is a finite collection of independent RBs (arms). At time t, the decision maker observes the state of all RBs, and chooses which RBs to activate. The decision maker then earns a reward that is the sum of the rewards of all RBs. This problem is called a multi-armed restless bandit. The name “restless” comes from the fact that an arm put at rest may still transition to a new state. Such a problem is notoriously difficult to solve, as its complexity grows exponentially with the number of arms. We next describe a very efficient heuristic to solve this problem, known as the Whittle index policy.

[2]It is also the first subcubic algorithm to compute Gittins index.

2.2 Indexability and Whittle index

Consider a RB process $(\boldsymbol{r}, \boldsymbol{P})$. For each $\lambda \in \mathbb{R}$, we define a λ-penalized RB whose transition matrix are the same as in the original RB and whose reward when taking action a in state i is $r_i^a - \lambda a$. The quantity λ can be viewed as a penalty for taking the active action. We consider that a decision maker seeks to maximize the average gain over an infinite horizon (the discounted case will be discussed in Section 7). To avoid the dependence on initial states, we assume that the RB is unichain[3].

For a given penalty λ, we denote by $\pi^*(\lambda)$ the minimal[4] policy that is optimal for a given penalty λ. When the penalty λ is very large, activating is highly penalized, in which case the optimal policy is to put all states to rest: $\pi = \emptyset$. At the opposite, when the penalty is very low, activating highly subsidized, in which case the optimal policy is to activate all states: $\pi = [n]$. A process is said to be indexable when the optimal policy $\pi^*(\lambda)$ is a strictly decreasing function of the penalty λ. More precisely, we define the indexability as follows:

Definition 2.1 (Indexability). An RB $(\boldsymbol{r}, \boldsymbol{P})$ is indexable if for all state $i \in [n]$, there exists a critical penalty λ_i such that the action "activate" is optimal in state i for the penalty λ if and only if $\lambda \leq \lambda_i$. In this case, we call λ_i the Whittle index of state i.

For an indexable RB, the minimal optimal policy for the penalty λ is the set of states whose index is strictly higher than λ: $\pi^*(\lambda) = \{i : \lambda < \lambda_i\}$. Note that the policy $\pi^*(\lambda)$ is not necessary the unique optimal policy. In particular, for a penalty equal to the Whittle index λ_i of state i both the action rest and activate are optimal in state i.

2.3 Characterization of indexability

For a given policy π and a given penalty λ, we denote by $g^\pi(\lambda)$ the average gain[5] of the policy π for the λ-penalized RB. We denote by $g^*(\lambda) = \max_{\pi \subset [n]} g^\pi(\lambda)$ the optimal average gain for the penalty λ.

Lemma 2.2. *A RB is indexable if and only if there is a non decreasing sequence of values $\mu^0 = -\infty < \mu^1 \leq \mu^2 \leq \cdots \leq \mu^n < \mu^{n+1} = +\infty$ and a sequence of policies $\pi^1 = [n] \supsetneq \pi^2 \supsetneq \cdots \supsetneq \pi^n = \emptyset$ such that π^k is an optimal policy for all penalty $\lambda \in [\mu^{k-1}, \mu^k]$ and the minimal optimal policy for all $\lambda \in [\mu^{k-1}, \mu^k)$ if $\mu^{k-1} \neq \mu^k$.*

Moreover, if the RB is indexable, then the index of state i is μ^k, where k is such that $i \in \pi^{k-1} \setminus \pi^k$.

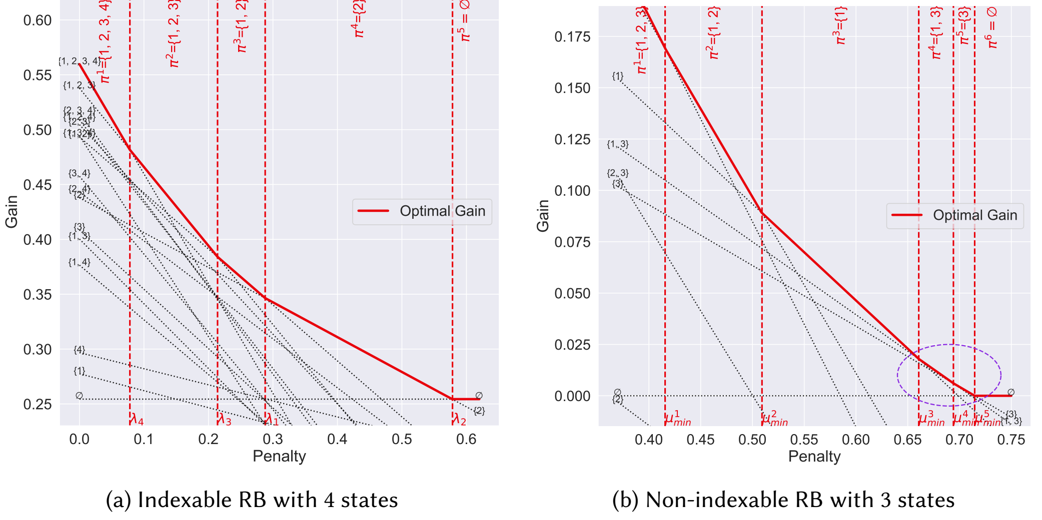
Proof. The lemma is a direct consequence of the definition of indexability. Indeed, assume first that there exist sequences μ and π that satisfy the condition of the lemma. For a state i, let $\lambda_i := \mu^k$, where k is such that $i \in \pi^{k-1} \setminus \pi^k$. Then, λ_i satisfies all conditions of the critical penalty of Definition 2.1. This implies that the problem is indexable. Assume now that the problem is indexable, and let σ^k be the state with the kth smallest index (where ties are broken arbitrarily). Let $\mu^k := \lambda_{\sigma^k}$ be the index of the state σ^k and define the policy $\pi^{k+1} = [n] \setminus \{\sigma^1, \ldots, \sigma^k\}$. Then the sequences μ^k and π^k satisfy the conditions of the lemma. □

To illustrate Lemma 2.2, we show in Figure 1 two RBs: one indexable (on the left) and one non-indexable (on the right). In both figures, we plot in dashed-dot black line the average gain $g^\pi(\lambda)$ as a function of λ for all policies $\pi \subset [n]$ (there are $2^4 = 16$ of them). We also plot in solid red the gain of the optimal policy. In Figure 1a), we observe that the sequence of optimal policies is decreasing: $\{1, 2, 3, 4\} \supset \{1, 2, 3\} \supset \{1, 2\} \supset \{2\} \supset \emptyset$, which means that this problem is indexable. The vertical

[3]a RB is unichain if all policies induce a Markov chain with a unique stationary distribution (see [41])

[4]Minimal for the inclusion order. Indeed, by Lemma A.1, if two policies π and π' are optimal for a given penalty λ, then the intersection $\pi \cap \pi'$ is also optimal. Hence, the minimal optimal policy is the intersection of all optimal policies.

[5]Note that as we assume that the RB is unichain, the average gain exists and does not depend on the state [41].



(a) Indexable RB with 4 states

(b) Non-indexable RB with 3 states

Fig. 1. Gain as a function of penalty for two examples, one which is indexable (Figure 1a) and other one which is not (Figure 1b).

red lines represent the indices of all states: for a given λ, the optimal policy (written in red) is to activate the process at all states whose λ_i is greater than the penalty λ. For Figure 1b, the sequence of optimal policies is $\{1,2,3\} \supset \{1,2\} \supset \{1\} \subset \{1,3\} \supset \{3\} \supset \emptyset$: this example is not indexable because of what happens around $\lambda \approx 0.7$: it is optimal to put 3 at rest when $\lambda \in [0.41, 0.66] \cup [0.71, +\infty]$ but not when $\lambda \in [0.66, 0.71]$.

3 CONDITION FOR INDEXABILITY AND WHITTLE INDEX COMPUTATION

This section aims at providing a basic algorithm to detect whether an arm is indexable or not and if it is the case, compute the Whittle index of any state i of the arm.

3.1 Overview of the Algorithm

Our algorithm computes Whittle index in increasing order by navigating through the optimal gain and using the characterization provided by Lemma 2.2. It follows the graphical construction given in Figure 1a. It also uses the following facts:

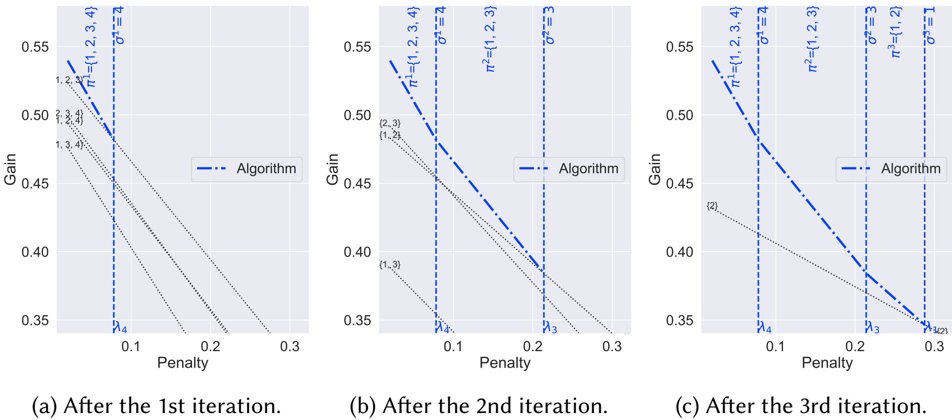
- For a given policy π, the gain $g^{\pi}(\lambda)$ is an affine function of λ. As a result, it is possible to compute the penalty λ at which two curves $g^{\pi}(\lambda)$ and $g^{\pi'}(\lambda)$ intersect.
- For an indexable RB, computing the index can be done by a greedy algorithm that constructs a sequence of penalties $\mu^1_{\min} \le \mu^2_{\min} \le \cdots \le \mu^n_{\min}$ and a sequence of policies $\pi^1 \supsetneq \cdots \supsetneq \pi^n$ by looking at where $g^{\pi^k}(\lambda)$ and $g^{\pi^k \setminus \{i\}}(\lambda)$ intersect for all $i \in \pi^k$.
- The RB is indexable if and only if for all k, π^k is an optimal policy for penalty μ^k.

This leads to Algorithm 1, that we write in pseudo-code. This algorithm relies on two subroutines: on Line 4, to compute the next index and on Line 8 to test if a policy is optimal. We will describe later in the paper how to implement these functions in an efficient manner. Note that in all the paper, we use the superscript k (*e.g.*, π^k or μ^k) to refer to the quantities computed at iteration k. We use the subscripts i or j (*e.g.*, $\pi_i, \lambda_i, \mu_i, \pi_j$) to refer to the quantities related to states i or j.

Algorithm 1: Test indexability and compute Whittle index if the problem is indexable.

1 Set $\pi^1 := [n], k := 1$, and $\mu^0_{\min} := -\infty$
2 **for** $k = 1$ *to* n **do**
3   **for** $i \in \pi^k$ **do**
4   Let $\mu^k_i = \inf\{\lambda : g^{\pi^k}(\lambda) = g^{\pi^k \setminus \{i\}}(\lambda)\}$
5   **if** $\arg\min_{i \in \pi^k : \mu^k_i \geq \mu^{k-1}_{\min}} \mu^k_i = \varnothing$ **then**
6   **return** *The problem is not indexable*
7   Let $\sigma^k := \arg\min_{i \in \pi^k : \mu^k_i \geq \mu^{k-1}_{\min}} \mu^k_i$ and $\lambda_{\sigma^k} := \mu^k_{\min} := \mu^k_{\sigma^k}$
8   **if** π^k *is not an optimal policy for the penalty* $\mu^k_{\min}$ **then**
9   **return** *The problem is not indexable*
10   Set $\pi^{k+1} := \pi^k \setminus \{\sigma^k\}$
11 **return** *The problem is indexable and the indices are* λ.

To illustrate how the algorithm works, we plot the first three iterations in Figure 2 for the RB represented in Figure 1a. The optimal policy for small λ is $\pi = [n]$ so the first iteration starts with $\pi^1 = \{1, 2, 3, 4\}$ and computes the equation of the affine function $g^{\pi^1}(\cdot)$. We then compute the equations of the affine functions $g^{\pi^1 \setminus \{i\}}(\cdot)$ for $i \in \pi^1$. This is represented in Figure 2a. We observe that $g^{\pi^1 \setminus \{4\}}(\cdot)$ is the first affine function to cross $g^{\pi^1}(\cdot)$ at $\mu^1_{min} = \lambda_4 \approx 0.9$. After that, we set $\pi^2 = \{1, 2, 3\}$ and compute the equation of the affine functions $g^{\pi^2 \setminus \{i\}}(\cdot)$ for $i \in \pi^2$ and use them to compute $\mu^2_{min} = \lambda_3 \approx 0.21$ (Figure 2b). We then continue with the affine functions $g^{\{2\}}(\cdot)$ and $g^{\{1\}}(\cdot)$ (note that $g^{\{1\}}(\cdot)$ is not visible on the figure because $g^{\{1\}}(\lambda) < 0.35$ for $\lambda \in [0, 0.3]$, see Figure 1a). We find $\mu^3_{min} = \lambda_1 \approx 0.28$ (Figure 2c). We are left with the last iteration (not shown here) to compute the intersection between $g^{\{2\}}(\lambda)$ and $g^{\varnothing}(\lambda)$ which occurs at $\mu^4_{min} = \lambda_2 \approx 0.57$ (see Figure 1a).



(a) After the 1st iteration. (b) After the 2nd iteration. (c) After the 3rd iteration.

Fig. 2. Illustration of the behavior of our algorithm for an arm with 4 states. We plot the average gain as a function of penalty λ of the various policy and of the policy computed by the algorithm (the model is the same as the one of Figure 1a.)

3.2 Correctness of Algorithm 1

The following result shows that Algorithm 1 is correct when testing indexability or computing Whittle index.

Theorem 1. *For a unichain RB, if the RB is indexable, then the vector λ returned by Algorithm 1 corresponds to the Whittle indices. If the RB is not indexable, then the algorithm returns that the problem is not indexable.*

Proof. Recall that $g^*(\lambda) = \max_\pi g^\pi(\lambda)$ is the optimal gain for penalty λ (*i.e.*, the red curve of Figure 1). We define the function $g^{algo}(\cdot)$ as the gain of the policies computed by Algorithm 1 (*i.e.*, the blue curve on Figure 2) as follows: For all $k \in \{1, \dots, n\}$ and $\lambda \in [\mu^{k-1}_{\min}, \mu^k_{\min}]$: $g^{algo}(\lambda) := g^{\pi^k}(\lambda)$. Our proof relies on the following lemma.

Lemma 3.1. *The three following facts are equivalent:*

(i) For all k: $g^*(\mu^k_{\min}) = g^{algo}(\mu^k_{\min})$
(ii) For all λ: $g^*(\lambda) = g^{algo}(\lambda)$.
(iii) The RB is indexable.

Indeed, assume that Lemma 3.1 holds. Then, if the RB is not indexable Line 5 or 8 will return "non-indexable" because of (i). If the RB is indexable, then the sequence of policies π^k satisfy the assumptions of Lemma 2.2 and the values λ_i are therefore the Whittle indices.

Proof of Lemma 3.1. (i) $\Leftrightarrow$ (ii): $g^{algo}(\cdot)$ is piecewise affine and convex. As such, it is the largest affine function that goes through the points $g^{algo}(\mu^k_{\min})$. Since g^* is the optimal gain, we have $g^{algo}(\lambda) \le g^*(\lambda)$. Since g^* is also convex and goes through $g^{algo}(\mu^k_{\min})$, both functions are equal.
(ii) $\Rightarrow$ (iii): If (ii) holds, then the sequence of policies π^k satisfies the assumptions of Lemma 2.2 and the values λ_i are therefore the Whittle indices.
(iii) $\Rightarrow$ (ii): Assume that the RB is indexable. We prove by induction on k that

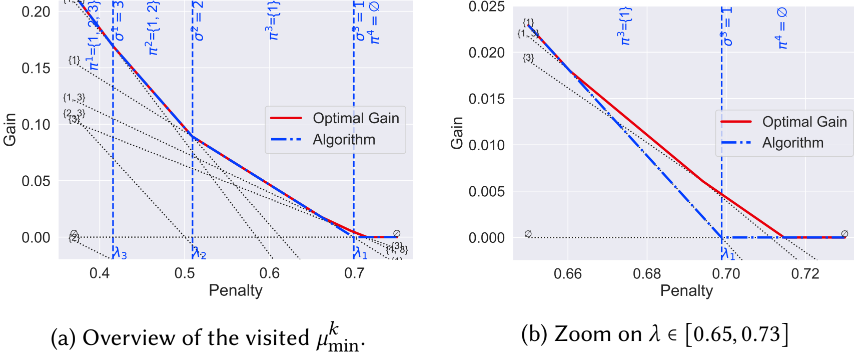
$$g^*(\lambda) = g^{algo}(\lambda) \qquad \text{for all } \lambda \le \mu^k_{\min}. \tag{1}$$

This is clearly true for $k = 0$ since $\mu^0_{\min} = -\infty$. Since $\mu^1_{\min}$ is the smallest penalty at which one of the curves $g^{\pi^1 \setminus \{i\}}(\cdot)$ crosses $g^{\pi^1}(\cdot)$, (1) also holds for $k = 1$.

Assume that (1) holds for some $k \ge 0$. If $\mu^{k+1}_{\min} = \mu^k_{\min}$, then (1) holds for $k+1$. Assume now that $\mu^{k+1}_{\min} > \mu^k_{\min}$. We argue that π^{k+1} is the minimal optimal policy for the penalty $\mu^k_{\min}$. Indeed, if there exists another policy $\pi' \subsetneq \pi^{k+1}$ that is also optimal, then by Lemma A.1, there would exist a policy $\pi^{k+1} \setminus \{i\}$ that would be optimal which would imply that $\mu^{k+1}_{\min} = \mu^k_{\min}$. Hence, we can assume that π^{k+1} is the minimal optimal policy for the penalty $\mu^k_{\min}$. Now, let $\nu = \inf\{\lambda > \mu^k_{\min}$ such that π^{k+1} is not optimal$\}$. By continuity of the functions $g^\pi(\lambda)$, there are two distinct optimal policies at ν (π^{k+1} and another policy π'). By definition of indexability, $\pi^{k+1} \supsetneq \pi'$. By Lemma A.1, this implies that there exists $\pi^{k+1} \setminus \{i\}$ that is optimal at ν, which implies that $\mu^{k+1}_{\min} = \nu$. □

We should note that

- in Line 4, μ^k_i can be infinite if there exists no penalty λ such that $g^{\pi^k}(\lambda) = g^{\pi^k \setminus \{i\}}(\lambda)$ but this case is still covered in the lines that follow.
- if $\mu^k_i < \mu^{k-1}_{\min}$ for all $i \in \pi^k$, then π^k remains optimal for all $\lambda > \mu^{k-1}_{\min}$. This is not possible for indexable problems because for an infinite penalty the only optimal strategy is to put all states at rest. That is why Line 5 returns that the problem is not indexable



(a) Overview of the visited $\mu_{\min}^k$.

(b) Zoom on $\lambda \in [0.65, 0.73]$

Fig. 3. Algorithm v.s. optimal gain for a non-indexable RB: We plot the average gain as a function of penalty for the non-indexable arm with 3 states of Figure 1b.

- by Line 7, it is possible to have $\mu_{\min}^k = \mu_{\min}^{k-1}$. This happens when several states have the same value of Whittle index. This is not problematic because we are sure that $\sigma^k \neq \sigma^{k-1}$ by Line 10.
- if $\mu_{\min}^k = +\infty$ then π^k cannot be optimal for $\mu_{\min}^k$ because, again, the only optimal policy for penalty $\lambda = +\infty$ is to put all states at rest, $\pi^*(\infty) = \emptyset$. Thus, Line 8 returns the problem is not indexable

□

3.3 Naive implementation of Algorithm 1 (in $O(n^4)$)

For a given penalty λ and a policy π, let $g^\pi(\lambda)$ be a scalar (called the average gain), and let $\boldsymbol{h}^\pi(\lambda)$ be a vector (called the bias) such that $h_1^\pi(\lambda) = 0$, and for all state $i \in [n]$:

$$g^\pi(\lambda) + h_i^\pi(\lambda) = r_i^{\pi_i} - \lambda\pi_i + \sum_{j=1}^{n} P_{ij}^{\pi_i} h_j^\pi(\lambda). \tag{2}$$

The above system is a system of $n+1$ linear equations with $n+1$ variables (the additional equation begins $h_1^\pi(\lambda) = 0$). As the RB is unichain, according to [41], the gain and bias are uniquely defined by the system of linear equations (2), together with the condition that $h_1^\pi(\lambda) = 0$. Note that in (2) the sum is for $j = 1$ to n. Since $h_1^\pi(\lambda) = 0$, it can be transformed into a sum from $j = 2$ to n.

Let us define the vector $\boldsymbol{v}^\pi(\lambda) := [g^\pi(\lambda), h_2^\pi(\lambda), \dots, h_n^\pi(\lambda)]^T$ which is similar to the vector $\boldsymbol{h}(\lambda)$ in which we replaced $h_1^\pi(\lambda)$ by $g^\pi(\lambda)$. We can write Equation (2) under a matrix form as:

$$A^\pi \boldsymbol{v}^\pi(\lambda) = \boldsymbol{r}^\pi - \lambda\boldsymbol{\pi}, \tag{3}$$

where $\boldsymbol{r}^\pi$ is the reward vector of policy π: $\boldsymbol{r}^\pi = [r_1^{\pi_1}, \dots, r_n^{\pi_n}]^T$, $\boldsymbol{\pi} = [\pi_1, \dots, \pi_n]^T$, and A^π is the following square matrix:

$$A^\pi := \begin{bmatrix} 1 & & & \\ 1 & 1 & & \\ 1 & & \ddots & \\ 1 & & & 1 \end{bmatrix} - \begin{bmatrix} 0 & P_{12}^{\pi_1} & \dots & P_{1n}^{\pi_1} \\ 0 & P_{22}^{\pi_2} & \dots & P_{2n}^{\pi_2} \\ & \vdots & & \vdots \\ 0 & P_{n2}^{\pi_n} & \dots & P_{nn}^{\pi_n} \end{bmatrix} = \begin{bmatrix} 1 & -P_{12}^{\pi_1} & \dots & -P_{1n}^{\pi_1} \\ 1 & 1-P_{22}^{\pi_2} & \dots & -P_{2n}^{\pi_2} \\ \vdots & & & \\ 1 & -P_{n2}^{\pi_n} & \dots & 1-P_{nn}^{\pi_n} \end{bmatrix} \tag{4}$$

As we show in Lemma A.2, in unichain MDP, the matrix A^π is invertible for any policy π. In consequence, $\boldsymbol{v}^\pi$ is an affine function of λ:

$$\boldsymbol{v}^\pi(\lambda) = (A^\pi)^{-1}(\boldsymbol{r}^\pi - \lambda\boldsymbol{\pi}) = (A^\pi)^{-1}\boldsymbol{r}^\pi - \lambda(A^\pi)^{-1}\boldsymbol{\pi} \tag{5}$$

The following lemma gives a characterization of the indifference between activating or resting a state under a given penalty. Its proof is in Appendix A.3.

Lemma 3.2. *Let π be a policy, $i \in \pi$ be any active state and λ be a penalty. Then the two policies π and $\pi \setminus \{i\}$ have the same gain for the penalty λ* (i.e., $g^{\pi}(\lambda) = g^{\pi\setminus\{i\}}(\lambda)$) *if and only if*

$$\delta_i - \lambda + \Delta_i v^{\pi}(\lambda) = 0, \tag{6}$$

where $\delta_i = r_i^1 - r_i^0$ and Δ_i is the vector such that $\Delta_{i1} = 0$ and $\Delta_{ij} = P_{ij}^1 - P_{ij}^0$ for $j \in \{2 \dots n\}$.

Using this result, computing a value μ_i^k at Line 4 of Algorithm 1 can be constructed by finding a penalty that satisfies (6) for the policy π^k, which is a (one-dimensional) linear equation. Indeed, let $\boldsymbol{d}^{\pi^k} = -(A^{\pi^k})^{-1}\boldsymbol{\pi}^k$. By (5), $\boldsymbol{v}^{\pi^k}(\lambda)$ is a linear function of λ whose derivative is $\boldsymbol{d}^{\pi^k}$. In particular, one has: $\boldsymbol{v}^{\pi^k}(\lambda) = \boldsymbol{v}^{\pi^k}(\mu_{\min}^{k-1}) + (\lambda - \mu_{\min}^{k-1})\boldsymbol{d}^{\pi^k}$. Using the lemma above for μ_{min}^{k-1}, we have $\boldsymbol{v}^{\pi^k}(\mu_{\min}^{k-1}) = \boldsymbol{v}^{\pi^{k-1}}(\mu_{\min}^{k-1})$. Hence, Equation (6) can be written as $\delta_i - \lambda + \Delta_i(\boldsymbol{v}^{\pi^{k-1}}(\mu_{\min}^{k-1}) + (\lambda - \mu_{\min}^{k-1})\boldsymbol{d}_i^{\pi^k}) = 0$ which is satisfied if and only if $\lambda = \mu_i^k$, where

$$\mu_i^k := \frac{\delta_i + \Delta_i \boldsymbol{v}^{\pi^{k-1}}(\mu_{\min}^{k-1}) - \mu_{\min}^{k-1}\Delta_i \boldsymbol{d}^{\pi^k}}{1 - \Delta_i \boldsymbol{d}^{\pi^k}}. \tag{7}$$

This shows that, for a given k, computing all μ_i^k of Line 4 can be done in $O(n^3)$: A $O(n^3)$ to inverse the matrix A^{π} and compute $\boldsymbol{d}^{\pi}$, plus some smaller order terms to compute the solutions of (7). Similarly, the test in Line 8 of Algorithm 1 can also be implemented in $O(n^3)$ by using Lemma 3.3 below. This leads to an overall complexity of $O(n^4)$ for Algorithm 1 that contains n loops each having a $O(n^3)$ complexity.

Lemma 3.3. *Let π be a policy, λ be a penalty and $\boldsymbol{v}^{\pi}(\lambda)$ as in* (5)*. Then the policy π is optimal for the penalty λ if and only if for all state i:*

$$\delta_i - \lambda + \Delta_i \boldsymbol{v}^{\pi}(\lambda) \leq 0 \text{ if } i \notin \pi \qquad \text{and} \qquad \delta_i - \lambda + \Delta_i \boldsymbol{v}^{\pi}(\lambda) \geq 0 \text{ if } i \in \pi \tag{8}$$

where δ_i and Δ_i are as in Lemma 3.2.

Proof. The proof is a direct consequence of *Policy Improvement* as described in Theorem 8.4.4 of [41]: a policy that satisfies (8) cannot be improved and is therefore optimal. □

4 COMPUTE WHITTLE INDEX IN $(2/3)n^3 + o(n^3)$

This section describes a way to implement Algorithm 1 efficiently using $O(n^3)$ operations. The main idea is to use the Sherman-Morrison formula to compute in $O(n^2)$ the bias vector associated to $g^{\pi\setminus\{i\}}(\lambda)$ from the one associated to $g^{\pi}(\lambda)$. This leads to a $O(n^3)$ algorithm. Once this main idea is in place, we show how to avoid unnecessary computations to obtain an algorithm that performs $(2/3)n^3 + o(n^3)$ arithmetic operations.

4.1 Additional Notations

In order to obtain a more efficient and compact algorithm, for an iteration k and a state i, we define $y_i^k := \Delta_i \boldsymbol{d}^{\pi^k}$ and $z_i^k := \Delta_i \boldsymbol{v}^{\pi^k}(\mu_{\min}^k)$, where $\boldsymbol{d}^{\pi^k}$, Δ_i and $\boldsymbol{v}^{\pi^k}$ are as in (7). Equation (7) can be rewritten as

$$\mu_i^k = \frac{\delta_i + z_i^{k-1} - \mu_{\min}^{k-1} y_i^k}{1 - y_i^k}. \tag{9}$$

The above equation can be used to compute μ_i^k and $\mu_{\min}^k$ easily from y_i^k and z_i^{k-1}.

For a given policy π, $\boldsymbol{v}^\pi$ is a linear function of λ whose derivative is $\boldsymbol{d}^\pi$. Hence, the value z_i^k can be computed from y_i^k and z_i^{k-1} as:

$$\begin{aligned} z_i^k &= \Delta_i \boldsymbol{v}^{\pi^k}(\mu_{\min}^k) \\ &= \Delta_i \boldsymbol{v}^{\pi^k}(\mu_{\min}^{k-1}) + \Delta_i \boldsymbol{d}^{\pi^k}(\mu_{\min}^k - \mu_{\min}^{k-1}) \\ &= z_i^{k-1} + (\mu_{\min}^k - \mu_{\min}^{k-1}) y_i^k \end{aligned} \tag{10}$$

where the third equality is true by Lemma 3.2. This shows that the critical values to compute are the variables y_i^k. In the remainder of this section, we show that the quantity y_i^k can be computed efficiently by a recursive formula.

4.2 Application of Sherman-Morrison Formula

To compute $y_i^{k+1} = \Delta_i \boldsymbol{d}^{\pi^{k+1}}$, we need to compute the quantities $\boldsymbol{d}^{\pi^{k+1}} = -(A^{\pi^{k+1}})^{-1}\boldsymbol{\pi}^{k+1}$. This requires the inverse of $A^{\pi^{k+1}}$. From the definition of π^k, two policies π^k and π^{k+1} differs by exactly one state: $\boldsymbol{\pi}^{k+1} = \boldsymbol{\pi}^k - \boldsymbol{e}_{\sigma^k}$. Also by definition of A^π in (4), the two matrices A^{π^k} and $A^{\pi^{k+1}}$ differs only at the line σ^k:

$$A^{\pi^{k+1}} = A^{\pi^k} + \boldsymbol{e}_{\sigma^k}\Delta_{\sigma^k} \tag{11}$$

where Δ_{σ^k} is a row vector defined as in the previous section.

One can compute efficiently the inverse of matrix $A^{\pi^{k+1}}$ from the one of A^{π^k} by using Sherman-Morrison formula, which says that if $A \in \mathbb{R}^{n\times n}$ is an invertible square matrix and $\boldsymbol{p}, \boldsymbol{q} \in \mathbb{R}^n$ are two column vectors such that $A + \boldsymbol{p}\boldsymbol{q}^\top$ is invertible, then:

$$(A + \boldsymbol{p}\boldsymbol{q}^\top)^{-1} = A^{-1} - \frac{A^{-1}\boldsymbol{p}\boldsymbol{q}^\top A^{-1}}{1 + \boldsymbol{q}^\top A^{-1}\boldsymbol{p}}.$$

Let $X_{ij}^k = \Delta_i (A^{\pi^k})^{-1}\boldsymbol{e}_j$. Following (11), we can apply Sherman-Morrison formula with matrix A^{π^k}; and vectors $\boldsymbol{p} = \boldsymbol{e}_{\sigma^k}$, and $\boldsymbol{q}^T = \Delta_{\sigma^k}$. After some simplification, we get:

$$\begin{aligned} X_{ij}^{k+1} &= \Delta_i (A^{\pi^{k+1}})^{-1}\boldsymbol{e}_j = \Delta_i (A^{\pi^k} + \boldsymbol{e}_{\sigma^k}\Delta_{\sigma^k})^{-1}\boldsymbol{e}_j \\ &= X_{ij}^k - \frac{X_{i\sigma^k}^k X_{\sigma^k j}^k}{1 + X_{\sigma^k\sigma^k}^k}. \end{aligned} \tag{12}$$

For $y_i^{k+1} = \Delta_i \boldsymbol{d}^{\pi^{k+1}} = -\Delta_i (A^{\pi^{k+1}})^{-1}\boldsymbol{\pi}^{k+1}$, we use the fact that $\boldsymbol{\pi}^{k+1} = \boldsymbol{\pi}^k - \boldsymbol{e}_{\sigma^k}$ and also apply Sherman-Morrison formula:

$$\begin{aligned} y_i^{k+1} &= -\Delta_i (A^{\pi^k} + \boldsymbol{e}_{\sigma^k}\Delta_{\sigma^k})^{-1}(\boldsymbol{\pi}^k - \boldsymbol{e}_{\sigma^k}) \\ &= -\Delta_i (A^{\pi^k})^{-1}(\boldsymbol{\pi}^k - \boldsymbol{e}_{\sigma^k}) + \frac{\Delta_i (A^{\pi^k})^{-1}\boldsymbol{e}_{\sigma^k}\Delta_{\sigma^k}(A^{\pi^k})^{-1}}{1 + \Delta_{\sigma^k}(A^{\pi^k})^{-1}\boldsymbol{e}_{\sigma^k}}(\boldsymbol{\pi}_k - \boldsymbol{e}_{\sigma^k}) \\ &= y_i^k + X_{i\sigma^k}^k + \frac{X_{i\sigma^k}^k(-y_{\sigma^k}^k - X_{\sigma^k\sigma^k}^k)}{1 + X_{\sigma^k\sigma^k}^k} \\ &= y_i^k + \frac{X_{i\sigma^k}^k(1 - y_{\sigma^k}^k)}{1 + X_{\sigma^k\sigma^k}^k} = y_i^k + (1 - y_{\sigma^k}^k)X_{i\sigma^k}^{k+1} \end{aligned} \tag{13}$$

The above formula indicate how to compute $\boldsymbol{y}^{k+1}$ from $\boldsymbol{y}^k$. To complete this analysis, let us show that $\boldsymbol{y}^1 = \boldsymbol{0}$. For a given policy π, the vector $\boldsymbol{d}^\pi$ satisfies the same equation as Equation (3) but

replacing $r_i^{\pi_i} - \lambda\pi_i$ by $-\pi_i$. This implies that for $\pi = \pi^1 = [1,\dots,1]^T$, one has $\boldsymbol{d}^\pi = [-1,0,\dots,0]^T$ as d_1^π is the average gain of a MDP whose reward is one in all states and $d_2^\pi,\dots,d_n^\pi$ is the bias of this MDP. This shows that for all i, we have $y_i^1 = \Delta_i \boldsymbol{d}^{\pi^1} = -\Delta_{i1} = 0$.

4.3 Detailed algorithm

Equation (9) shows how to compute μ_i^k from the values of y_i^k and z_i^{k-1} while (13), (12) and (10) show how to compute the values of y, z and X recursively in k. In order to compute $\mu_{\min}^k$ and σ^k, one only needs to compute the values μ_i^k for $i \in \pi^k$. Once $\mu_{\min}^k$ is determined as in Line 7, we need to verify the optimality of π^k for penalty $\mu_{\min}^k$. That is, we check whether for any passive state $i \notin \pi^k$, the rest action is worst than the activate action in state i for penalty $\mu_{\min}^k$. If it is the case, then the process is non-indexable. By Lemma 3.3, to do so, it is sufficient to check that for $i \in [n] \setminus \pi^k$:

$$\mu_{\min}^k < \delta_i + \Delta_i \boldsymbol{v}^{\pi^k}(\mu_{\min}^k) = \delta_i + z_i^k. \tag{14}$$

Algorithm 2: Test indexability and compute Whittle indices (if indexable).

1 **Init.**
2 Set $\pi^1 = [n]$, $k_0 = 1$, $\delta_i = r_i^1 - r_i^0$, $\Delta_{i1} = 0$ and $\Delta_{ij} = P_{ij}^1 - P_{ij}^0, \forall i \in [n], j \in \{2\dots n\}$.
3 Set $X^1 = \Delta(A^{\pi^1})^{-1}$, where A^{π^1} is defined by (4).
4 Set $\mu_{\min}^0 = -\infty$, $\boldsymbol{y}^1 = \boldsymbol{0}$, and $\boldsymbol{z}^0 = X^1 \boldsymbol{r}^{\pi^1}$
5 **for** $k = 1$ ***to*** n **do**
6 **for** $i \in \pi^k$ **do**
7 Set $\mu_i^k = \dfrac{\delta_i + z_i^{k-1} - \mu_{\min}^{k-1} y_i^k}{1 - y_i^k}$ *# For* $k = 1$*:* $\mu_{\min}^0 y_i^1 = -\infty \times 0 = 0$.
8 **if** $\arg\min_{i \in \pi^k : \mu_i^k \geq \mu_{\min}^{k-1}} \mu_i^k = \varnothing$ **then**
9 **return** *The problem is not indexable*
10 Set $\sigma^k := \arg\min_{i \in \pi^k : \mu_i^k \geq \mu_{\min}^{k-1}} \mu_i^k$ and $\lambda_{\sigma^k} = \mu_{\min}^k = \mu_{\sigma^k}^k$
11 Set $\boldsymbol{z}^k = \boldsymbol{z}^{k-1} + (\mu_{\min}^k - \mu_{\min}^{k-1})\boldsymbol{y}^k$ *# For* $k = 1$*, one has* $\boldsymbol{z}^1 = \boldsymbol{z}^0$.
12 **for** $i \in [n] \setminus \pi^k$ **do**
13 **if** $\mu_{\min}^k < \delta_i + z_i^k$ **then**
14 **return** *The problem is not indexable*
15 **if** $k < n$ **then**
16 Set $\pi^{k+1} = \pi^k \setminus \{\sigma^k\}$
17 Update_X(k) *# Here we call Subroutine 3 or Subroutine 4*
18 Set $\boldsymbol{y}^{k+1} = \boldsymbol{y}^k + (1 - y_{\sigma^k}^k) X_{:\sigma^k}^{k+1}$
19 **return** *The problem is indexable and the indices are* λ.

This leads to Algorithm 2 that can be decomposed as follows:

(1) In Line 2 to 4, we initialize the various variables. The main complexity of this part is to compute the matrix X^1, which is equivalent to solving the linear system $XA^{\pi^1} = \Delta$. It can be done by inverting the matrix A^{π^1} and multiplying this by the matrix Δ. This can be done in a subcubic complexity by using for instance Strassen's algorithm [44].

(2) We then enter the main loop:

Subroutine 3: Update_X()

```
1 for ℓ = 1 to k do
2     for i ∈ [n] do                         # or i ∈ π^{ℓ+1} if we do not test indexability.
3         X^{ℓ+1}_{iσ^k} := X^ℓ_{iσ^k} − X^ℓ_{iσ^ℓ} / (1 + X^ℓ_{σ^ℓσ^ℓ}) · X^ℓ_{σ^ℓσ^k}
```

- We update the vectors μ, $\boldsymbol{z}$ by using Equations (9) and (10), and test indexability. This costs $O(n)$ operations per iteration, thus $O(n^2)$ in total.
- We update the vector X according to (12). The "naive" way to do so is to use Subroutine 3. At iteration k this costs $2kn$ arithmetic operations if we want to test indexability, and $2\sum_{l=1}^{k}(n-l)$ if we do not want to test indexability. The total complexity of computing X is $n^3+O(n^2)$ arithmetic operations if we test indexability and $(2/3)n^3+O(n^2)$ if we do not. A detailed study of the arithmetic complexity is provided in Appendix B, where we also provide details on how to efficiently implement the algorithm, including how to optimize memory costs.
- In Line 18, we update the vector $\boldsymbol{y}$ by using Equations (13), which costs $O(n)$ per iteration.

Hence, the total complexity of this algorithm is $n^3+o(n^3)$ if we test indexability and $(2/3)n^3+o(n^3)$ if we do not test indexability. Without testing the indexability, our algorithm has the same main complexity as [38]. However, the algorithm of [38] computes Whittle index only for an arm that is PCL-indexable. Hence we can claim our algorithm is the first algorithm that computes Whittle index with cubic complexity for all indexable restless bandits. It is also the first algorithm that detects non-indexability of restless bandits.

Remark: Equivalently, instead of calling Subroutine3 at Line 17, one could do the following update for all $j \in \pi^k$ and $i \in [n]$ (or $i \in \pi^k$ if we do not test indexability):

$$X^{k+1}_{ij} := X^k_{ij} - \frac{X^k_{i\sigma^k}}{1+X^k_{\sigma^k\sigma^k}} X^k_{\sigma^k j}. \tag{15}$$

This iterative update is very close to the one used in [5, 38] for discounted restless bandit. This results in an algorithm that has the same total complexity as Subroutine 3 (of $n^3+O(n^2)$ or $(2/3)n^3+O(n^2)$ with or without the indexability test) because both algorithms will have computed the same values of X^k_{ij}. The reason to use Subroutine 3 is that, as we will see in the next section, not all values of X^k_{ij} are needed: in particular, for $\ell < k$, the computation of the value $X^\ell_{i\sigma^k}$ has no interest per say and is only useful because it allows to recursively compute $X^{k+1}_{i\sigma^k}$. In the section below, we show how to reduce the cost by avoiding the computation of $X^\ell_{i\sigma^k}$ when ℓ is much smaller than k. We comment more on the differences with [5, 38] in Appendix D and in particular we explain why our approach can be tuned into a subcubic algorithm while (15) cannot.

5 A SUBCUBIC ALGORITHM

5.1 Main idea: recomputing X^k from A^{π^k} periodically

The main computational burden of Algorithm 2 is concentrated on two lines: on Line 3 where we compute X^1 by solving a linear system, and on Line 17 where we compute $X^{k+1}_{:\sigma^k}$ from $X^1_{:\sigma^k}$. The remainder of the code runs in $O(n^2)$ operations and is therefore negligible for large matrices. In fact, the computation of X^1 is done by solving a linear system, which can be computed by using a

subcubic algorithm (like Strassen [44]). In this section, we show how to optimize our algorithm by reducing the complexity of the update_X() function, at the price of recomputing the full matrix X^{k_0} from $A^{\pi^{k_0}}$ periodically.

At iteration k of Algorithm 2, the quantities $X^{k+1}_{i\sigma^k}$ are used at Line 18 to obtain the values $\boldsymbol{y}^{k+1}$. The matrix X^k is defined as $X^k = \Delta(A^{\pi^k})^{-1}$. It also satisfies Equation (12), that is, for an iteration ℓ and states i and j, we have:

$$X^{\ell+1}_{ij} = X^{\ell}_{ij} - \frac{X^{\ell}_{i\sigma^\ell} X^{\ell}_{\sigma^\ell j}}{1 + X^{\ell}_{\sigma^\ell \sigma^\ell}}. \tag{16}$$

The way Subroutine 3 is implemented is to initialize $X^1 = \Delta(A^{\pi^1})^{-1}$ and then use (16) recursively to compute the vector $X^{k+1}_{:\sigma^k}$ from $X^1_{:\sigma^k}$ at each iteration.

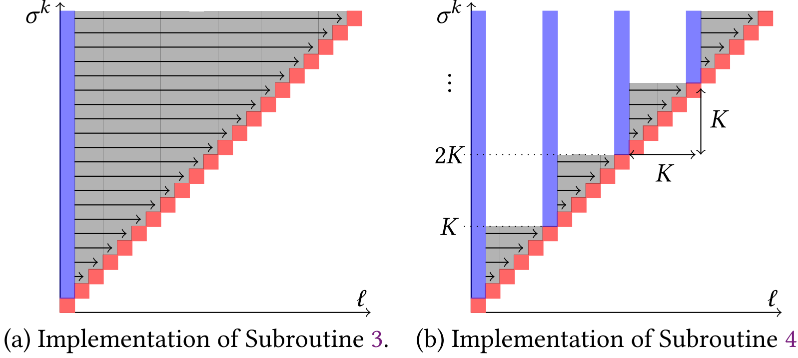
Here, we propose an alternative formulation which consists in recomputing the whole matrix $X^k = \Delta(A^{\pi^k})^{-1}$ every K iterations. In the meantime, we use (16) to compute the values $X^{\ell}_{i\sigma^{k-1}}$ for $\ell \in \{k_0+1, \dots, k\}$ where k_0 is the iteration at which we recomputed the whole matrix $X^{k_0} = \Delta(A^{\pi^{k_0}})^{-1}$. This can be implemented by changing the call to Subroutine 3 on Line 17 by a call to Subroutine 4.

Subroutine 4: Update_X_FMM(k)

1 **if** *k is an iteration at which we recompute the whole X^{k+1}* **then**
2 Set $k_0 := k + 1$
3 Set $X^{k_0} = \Delta(A^{\pi^{k_0}})^{-1}$
4 **for** $\ell = k_0$ *to* k **do**
5 **for** $i \in [n]$ **do** *# or $i \in \pi^{\ell+1}$ if we do not test indexability.*
6 $X^{\ell+1}_{i\sigma^k} := X^{\ell}_{i\sigma^k} - \dfrac{X^{\ell}_{i\sigma^\ell}}{1 + X^{\ell}_{\sigma^\ell\sigma^\ell}} X^{\ell}_{\sigma^\ell \sigma^k}$

To see why this can be more efficient, we illustrate in Figure 4 the pairs (ℓ, σ^k) for which we compute the vector $X^{\ell}_{:\sigma^k}$, either for Subroutine 3 or Subroutine 4. In each case, a vertical blue line indicates that we recompute the whole matrix X^k by solving a linear system. The gray zone corresponds to the values (ℓ, σ^k) for which we compute $X^{\ell}_{:\sigma^k}$ using Equation (16) and the red squares represent the vector $X^{k+1}_{:\sigma^k}$ used at iteration k of Algorithm 2. For Subroutine 3, we do one matrix inversion at the beginning and then compute for all (ℓ, σ^k) with $\ell \le k$ because the red square at value $(k+1, k)$ is computed starting from the vertical blue line at value $(1, k)$. For Subroutine 4, we do n/K (here $n/K = 4$) full recomputation of X^k, which correspond to the vertical blue lines. We gain in terms of operations because the surface of the gray zone to compute is divided by n/K.

Note that the y-axis of Figure 4 is ordered by increasing value of σ^k (and not by increasing value of k). The value of σ^k is computed at iteration k but is unknown before. This explains why in Subroutine 4, when we recompute the matrix (X^{k+1}_{ij}) at an iteration k (vertical blue lines in Figure 7(b)), we recompute it for all i, j and not just $i, j \in \{\sigma^{k+1} \dots \sigma^K\}$ (which are the only values that we will use): Indeed, $\sigma^{k+1} \dots \sigma^K$ are not known at iteration k.



(a) Implementation of Subroutine 3. (b) Implementation of Subroutine 4

Fig. 4. Illustration of the improvement proposed by replacing Subroutine 3 by Subroutine 4. For Subroutine 4, we solve more linear systems (each vertical blue line corresponds to solving a linear system) but we reduce the gray zone to compute. A linear arrow corresponds to the internal loop of Line 4 of Subroutine 4.

5.2 A subcubic Algorithm for Whittle Index

We now assume to have access to a subcubic matrix multiplication algorithm that satisfies the following property:

(FMM) There exists an algorithm to multiply a matrix of size $n \times n$ by a matrix of size $n \times n^\alpha$ that runs in $O(n^{\omega(\alpha)})$, where $\omega : [0,1] \to [2,3]$ is a non-decreasing function.

The naive algorithm to compute the product of two square matrices of size $n \times n$ runs in $O(n^3)$. In the seminal work [44], Strassen showed that the exponent 3 is not optimal and derived an algorithm that runs in $O(n^{\log_2 7})$ – known as Strassen's algorithm. This shows that $\omega(\alpha) \leq \log_2 7 \approx 2.8$. Today, finding an algorithm with the smallest $\omega(\alpha)$ is an active area of research. For square matrices (i.e. $\alpha = 1$), the best known exponent today is $\omega(1) \leq 2.37286$ [8] and is based on a variant of Coppersmith-Winograd [18]. Note that it is known that matrix inversion is not harder than matrix multiplication [19], hence, computing the inverse of a square matrix can also be done in $O(n^{\omega(1)})$. A similar reduction exists for multiplying rectangular matrices. It is shown in [29, 39], that if there exists an algorithm to multiply a matrix of size $n \times n$ by a matrix of size $n \times n^\alpha$ that runs in $O(n^{\omega(\alpha)})$, then there also exists an algorithm in $O(n^{\omega(\alpha)})$ to multiply a matrix $n \times n^\alpha$ by a matrix of size $n^\alpha \times n$. The best upper bound on $\omega(\alpha)$ known today are given in [21].

Going back to Algorithm 2 where Line 17 is Subroutine 4, we now assume that we recompute the whole matrix X^k every $O(n^\alpha)$ iterations. The new algorithm has a subcubic complexity:

Theorem 5.1. *Algorithm 2 with Subroutine 4 checks indexability and computes Whittle (and Gittins) index in time at least $\Omega(n^{2.5})$ and at most $O(n^{2.5286})$ when choosing $\alpha = 0.5286$.*

We believe that Theorem 5.1 is the first result that shows that Whittle index can be computed in subcubic time. As we show in Section 7, this algorithm can be directly extended to discounted index. As a byproduct, we also obtain the first subcubic algorithm to compute Gittins index.

Proof. The algorithm starts by computing X^1 which can be done in $O(n^{\omega(1)})$. Then, there are $n^{1-\alpha}$ times that we do:

(1) We fill the "gray" mini matrices by using (16). This amounts to three for loops of size n (for i), n^α (for k) and n^α (for ℓ). Hence, each small gray matrix costs $O(n^{1+2\alpha})$.

(2) At the end of a cycle, we recompute the full inverse by updating $(A^{\pi^k})^{-1}$ from $(A^{\pi^{k-n^\alpha}})^{-1}$. As we show in Lemma 5.2 (stated below), this can be done in $O(n^{\omega(\alpha)})$.

This implies that the algorithm has a complexity:

$$O(n^{\omega(1)}) + n^{1-\alpha}\left(O(n^{1+2\alpha}) + O(n^{\omega(\alpha)})\right) = O(n^{\max\{2+\alpha, 1-\alpha+\omega(\alpha)\}}).$$

To compute the optimal value of α minimizing this expression requires the knowledge of the function $\omega(\alpha)$ which is not known. The current state of the art only gives a lower bound ($\omega(\alpha) \geq 2$) and an upper bound described in [21].

It is shown in [21] that $\alpha = 0.5286$ is the smallest currently known value of α for which $\omega(\alpha) < 1 + 2\alpha$. This implies that the complexity is at most $O(n^{2.5286})$.

As for the lower bound, $\omega(\alpha) \geq 2$ implies that the complexity of the algorithm is at least $\Omega(n^{2.5})$. □

In the next lemma, B plays the role of A^{π^k} and A the role of $A^{\pi^{k-n^\alpha}}$. Note that as required in the lemma, exactly n^α rows and columns are changed between the two.

LEMMA 5.2. *Assume (FMM). Let A be a square matrix whose inverse A^{-1} has already been computed, and let B be an invertible square matrix such that $A - B$ is of rank smaller than n^α. Then, it is possible to compute the inverse of B in $O(n^{\omega(\alpha)})$.*

PROOF. The matrix B can be written as $B = A + UCV$ where U is a $n \times n^\alpha$ matrix, C is $n^\alpha \times n^\alpha$ and V is $n^\alpha \times n$. The Sherman–Morrison–Woodbury formula [48] states that

$$B^{-1} = (A + UCV)^{-1} = A^{-1} - A^{-1}U\left(C^{-1} + VA^{-1}U\right)^{-1}VA^{-1}.$$

This shows that B^{-1} can be computed by:

- Computing $D := A^{-1}U$ and $E := VA^{-1}$: this takes $O(n^{\omega(\alpha)})$.
- Computing $F := \left(C^{-1} + VA^{-1}U\right)^{-1}$: as this is the inversion of a $n^\alpha \times n^\alpha$ matrix, it can be done in $O(n^{\alpha\omega(1)})$ where $\alpha\omega(1) \leq \omega(\alpha)$.
- Computing $G := DF$ and then GE: this again takes $O(n^{\omega(\alpha)})$.

Hence, computing B^{-1} can be done in $O(n^{\omega(\alpha)})$ operations for the inversion and all multiplications plus an additional $O(n^2)$ term for the subtraction and the addition. As $\omega(\alpha) \geq 2$, this concludes the proof of the lemma. □

5.3 Subcubic algorithm in practice

The complexity of $O(n^{2.5286})$ given in Theorem 5.1 is mainly of theoretical interest. The value $\alpha = 0.5286$ is obtained by using the best upper bound on $\omega(\alpha)$ known today which is based on the Coppersmith-Winograd algorithm and its variants. The Coppersmith-Winograd algorithm (or its variants) are, however, known as a *galactic algorithm*: the hidden constant in the $O()$ is so large that their runtime is prohibitive for any reasonable value of n. Hence, the existence of these algorithms is of theoretical interest but has limited applicability.

This does not discard the practical improvement provided by Subroutine 4 which is based on the mere fact that multiplying two matrices (or inverting a matrix) is faster than three nested loops even for matrices of moderate size. To verify this, we launched a detailed profiling of the code of Algorithm 2 with the non-optimized Subroutine 3. It shows that for a problem of dimensions 5000, the update of Line 17 takes more than 90% of the computation time, the initialization of X^1 on Line 3 takes about 5% of the time and the rest of the code takes less than 1% of the running time.

Now, if inverting the full matrix takes about 5% of the execution time, and updating the gray zone takes 95%, then by doing 5 updates, one can hope to obtain an algorithm whose running time is

roughly $5 \times 5 + 95/5 \approx 43\%$ the one of the original implementation. As we observe in Section 6, this is close to the gain that we obtain in practice. A general way to choose the best number of updates is used in the numerical section. It is based on the following reasoning. For large matrices (say $n \geq 10^3$), the fastest implementations of matrix multiplication and inversion are based on Strassen's algorithm [27, 28]. As we report in Appendix C, the time to solve a linear system of size n by using the default installation of `scipy` seems to run in $O(n^{2.8})$. By replacing the function $\omega(\alpha)$ used in Theorem 5.1 by a more practical bound ($\omega(\alpha) = 2.8$), the best value for α becomes $\alpha = 0.9$. This indicates that our algorithm can be implemented in $O(n^{2.9})$ by doing $O(n^{0.1})$ recomputation of X^k from A^{π^k}. Note that even for very large values of n (like $n = 15000$), $n^{0.1}$ remains quite small, *e.g.*, $15000^{0.1} \approx 2.6$. In practice, we observe that updating $\text{int}(2n^{0.1})$ times (the notation $\text{int}(x)$ indicates that it is rounded to the closest integer) gives the best performance among all algorithms, as reported in the next section.

6 NUMERICAL EXPERIMENTS

In complement to our theoretical analysis, we developed a python package that implements Algorithm 2 and gives the choice of using the variant of Subroutine 3 or of Subroutine 4 to do the "update_X()" function. This package relies on three python libraries: `scipy` and `numpy` for matrix operations, and `numba` to compile the python code. To facilitate its use, this package can be installed by using `pip install markovianbandit-pkg`.

All experiments were run on a laptop (Macbook Pro 2020) with an Intel Core i9 CPU at 2.3 GHz with 16GB of Memory using Python 3.6.9 :: Anaconda custom (64-bit) under macOS Big Sur version 11.6.2. The version of the packages are `scipy` version 1.5.4, `numpy` version 1.19.5 and `numba` version 0.53.1. The code of all experiments is available at https://gitlab.inria.fr/markovianbandit/efficient-whittle-index-computation.

6.1 Time to compute Whittle indices

To test the implementation of our algorithm, we randomly generate restless bandit instances with n states where $n \in \{100, 1000, \dots, 15000\}$. In each case, both transition matrices are uniform probabilistic matrices: for each row of each matrix, we generate n i.i.d. entries following the exponential distribution and divide the row by its sum. This means that all matrices are dense. Note that all tested matrices are indexable. This is coherent with [22, 36] that report that for uniform matrices, the probability of finding a non-indexable example decreases very rapidly with the dimension n. Finally, reward vectors were generated from random Uniform[0,1) entries.

We record the runtime of the different variants of our algorithm and report the results in Table 1. Note that these results present the whole execution time of the algorithm, including the initialization phase in which X^1 is computed. For each value of n, we run Algorithm 2 with four variants:

- The first two correspond to the $O(n^3)$ algorithm (that uses Subroutine 3 for Line 17), either (a) with the indexability test, or (b) without the indexability test.
- The next two correspond to the subcubic algorithm (that uses Subroutine 4 for Line 17 with $\text{int}(2n^{0.1})$ updates), either (c) with the indexability test, or (d) without the indexability test.

Our numbers show that our algorithm can compute the Whittle index in less than one second for $n = 1000$ states and slightly less than 7 minutes for $n = 15000$ states with variant (d). As expected, not doing the indexability test does improve the performance compared to doing the indexability test (here by a factor approximately 1/3 for Subroutine 3 and 1/5 for Subroutine 4). More importantly, this table shows that the time when using the subcubic variant, Subroutine 4, diminishes the computation time by about 40% to 50% compared to when using Subroutine 3. Note that for $n = 100$, using the Subroutine 3 is slightly faster than using the subroutine 4 (note that

Table 1. Running time (in seconds) of the variants of Algorithm 2

n	$O(n^3)$ algorithm (Subroutine 3) (a) With index. test	(b) Without test	Subcubic algorithm (Subroutine 4) (c) With index. test	(d) Without test
100	0.006	0.004	0.007	0.005
1000	0.2	0.2	0.2	0.2
2000	1.9	1.2	1.1	1.0
3000	7.2	5.0	3.2	2.6
4000	17	12	8	6
5000	34	25	16	12
6000	60	43	27	20
7000	95	68	42	33
8000	142	99	63	49
9000	199	141	92	70
10000	275	191	122	95
11000	361	257	164	133
12000	471	339	225	190
13000	620	428	286	243
14000	790	553	408	317
15000	965	685	501	403

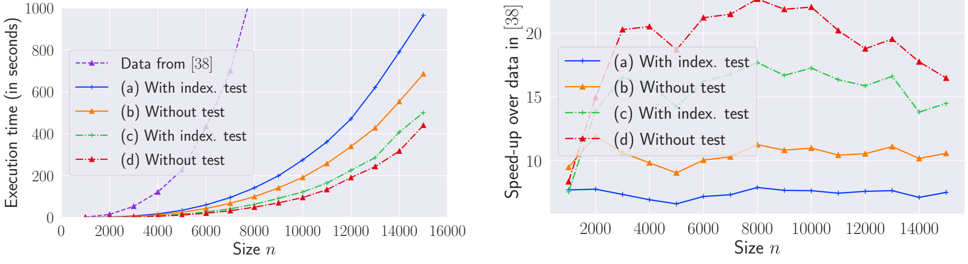
both takes only a few milliseconds). This indicates that the subcubic algorithm becomes interesting when n is large enough (say $n \geq 2000$).

To give a visual idea of how the various variants of the algorithms compare, we plot in Figure 5a the runtime of the four variants along with the runtime reported [38], that we believe was the fastest algorithm to compute Whittle index so far. We plot in Figure 5b the runtime of each variant divided by the runtime of the algorithm of [38]. For large n, our best implementation is about 10 to 20 times faster than the one of [38]. For instance, for $n = 15000$, our implementation takes about 7 minutes to compute the index (or 9 minutes when checking indexability) whereas [38] reports 2 hours. Recall that the algorithm of [38] cannot check indexability. In our implementation, not testing indexability reduces the computation time of 15 – 20% for Subroutine 4 or 25 – 30% for Subroutine 3 compared to the version that tests indexability.

It should be clear that the comparison between our implementation and the numbers reported in [38] has its limits. First, there is no public implementation of the algorithm of [38] and we did not re-implement it. Hence, the computation was not done on the same machine, and not implemented in the same programming language. The characteristic of the machine described in [38] seems comparable if not better to ours, and our implementation is in python/numpy whereas the one of [38] is in matlab. Second, it is written in [38] that the reported numbers do not "count the initialization stage of computing the initial tableau". There is no mention of the cost of the initialization stage in [38]. On the contrary, the numbers that we report in this paper include all initialization stage, which puts our running time at disadvantage.

6.2 Statistics of indexable problems

To the best of our knowledge, our algorithm provides the first indexability test that scales well with the dimension n. We used this to answer a very natural question: given a randomly generated bandit, how likely is it to be indexable? This question was partially answered in [36] that shows that when generating dense RBs, the probability of generating a non-indexable RB is close to 10^{-n}



(a) Runtime of our implementations as a function of the state size n.

(b) Speedup of each implementation over the data from [38] as a function of the state size n.

Fig. 5. Numerical result over 7 simulations taking around 21 hours in total: in each simulation, we run the algorithm over randomly generated RBs with the state size ranging over $\{1000, \ldots, 15000\}$. We plot the average runtime over 7 simulations. The solid lines represent the result of Subroutine 3 and the dashed-dot lines represent the one of Subroutine 4. The marker "+" indicates that algorithms test indexability and the triangles indicates that algorithms do not test indexability.

for $n \in \{3 \ldots 7\}$. This suggests that most RBs are indexable. Below, we answer two questions: what happens for larger values of n, and more importantly, what happens when the RBs are not dense?

To answer these questions, we consider randomly generated RBs where the matrices P^0 and P^1 are b-diagonal matrices with b non-null diagonals. In particular, $b = 3$ corresponds to tridiagonal matrices, $b = 5$ corresponds to pentadiagonal matrices and $b = 7$ corresponds to septadiagonal matrices. We also compare with the classical case of dense matrices (which corresponds to $b = 2n-1$). For each model, the entries are generated from the exponential distribution for each row and we divide the row by the sum of generated entries for this row. We vary n from 3 to 50 and for each case, we generate 100000 RBs. We report in Table 2 the number of indexable RBs for each. Note that pentadiagonal matrices are dense matrices for $n = 3$ and septadiagonal matrices are dense matrices for $n = 4$ and do not make sense for $n = 3$, which is why no numbers are reported.

Table 2. Number of indexable problems among 100 000 randomly generated problems.

n	Tridiagonal	5-diagonal	7-diagonal	Dense
3	98 731	–	–	99 883
4	95 067	99 655	–	99 931
5	89 198	99 309	99 902	99 969
10	54 129	90 377	98 914	100 000
30	7 094	29 699	66 143	100 000
50	1 823	9 332	32 069	100 000

Based on these results, we can assert that dense models are essentially always indexable which conforms with the data reported in [36]. The situation is, however, radically different for sparse models: the number of indexable problems decreases quickly with the number of states. For instance, there are only 1 823 indexable 50-state problems among 100 000 generated tridiagonal models (*i.e.* around 1.8% are indexable). Note that a tridiagonal model is a birth-death Markov chain which is frequently used for queueing systems. Hence, it is very important to check the indexability of the problem because it is not a prevalent property for sparse models. This also calls for new efficient policies in multi-arm bandits problems that are not based on Whittle indices.

7 EXTENSION TO THE DISCOUNTED CASE AND GITTINS INDEX

The model described in Section 2 corresponds to the definition of Whittle index for a time-average criterion, for which Whittle index is known to be asymptotically optimal [46] for restless bandits. Yet, Whittle index can also be defined for the discounted case [5, 38]. Notably, the discounted Whittle index simplifies into Gittins index when the bandit is rested (*i.e.*, when $P^0 = I$ and $\boldsymbol{r}^0 = 0$.) In this section, we show how to adapt our algorithm to the discounted case. As a by product, we obtain the first subcubic algorithm to compute Gittins index in the discounted case.

7.1 Discounted Whittle Index

We now consider a λ-penalized RB process in which the instantaneous reward received at time t is discounted by a factor β^t, where $\beta \in (0,1)$ is called the discount factor: when executing action a in state i at time t, the decision maker earns a reward $\beta^t(r_i^a - \lambda a)$. For a given policy π, we denote by $u_i^\pi(\lambda)$ the expected discounted reward earned by the decision maker when the RB starts in state i at time 0. The vector $\boldsymbol{u}^\pi(\lambda) = [u_1^\pi(\lambda) \dots u_n^\pi(\lambda)]^T$ is called the value function of the policy π. From [41], it satisfies Bellman's equation, that is, for all state i we have:

$$u_i^\pi(\lambda) = r_i^{\pi_i} - \lambda \pi_i + \beta \sum_{j=1}^{n} P_{ij}^{\pi_i} u_j^\pi(\lambda). \tag{17}$$

The above equation is a linear equation, whose solution is unique because $\beta < 1$. It is given by:

$$\boldsymbol{u}^\pi(\lambda) = (I - \beta P^\pi)^{-1}(\boldsymbol{r}^\pi - \lambda \boldsymbol{\pi}). \tag{18}$$

For a given penalty λ and a state i, we denote by $u_i^*(\lambda) = \max_\pi u_i^\pi(\lambda)$ be the optimal value of state i. A policy π is optimal for the penalty λ if for all state $i \in [n]$, $u_i^*(\lambda) = u_i^\pi(\lambda)$. By [41] such a policy exists. Similarly to the time-average criterion studied before, a β-discounted RB is called indexable if for all state $i \in [n]$, there exists a critical penalty λ_i such that the action "activate" is optimal in state i if and only if $\lambda_i \geq \lambda$.

7.2 Analogy between the time-average and the discounted versions

Let π be a policy and $i \in \pi$ be an active state. By Equation (17), the two actions "rest" and "activate" are equivalent in state i if and only if

$$r_i^1 - \lambda + \beta \sum_{j=1}^{n} P_{ij}^1 u_j^*(\lambda) = r_i^0 + \beta \sum_{j=1}^{n} P_{ij}^0 u_j^*(\lambda),$$

which is equivalent to

$$\lambda = \delta_i + \tilde{\Delta}_i \boldsymbol{u}^\pi(\lambda), \tag{19}$$

where δ_i is defined as for the time-average case and $\tilde{\Delta}$ is such that for all[6] states $i, j \in [n]$: $\tilde{\Delta}_{ij} := \beta(P_{ij}^1 - P_{ij}^0)$. This equation is the same as its time-average analogue (6), but replacing Δ by $\tilde{\Delta}$.

To finish the derivation of the algorithm, one should note that the value function $\boldsymbol{u}(\lambda)$ plays the same role as the vector $\boldsymbol{v}(\lambda)$ defined for the time-average case. In particular, the definition of $\boldsymbol{u}$ in Equation (18) is the analogue of the definition of $\boldsymbol{v}$ in (5) up to the replacement of the matrix A^π in (5) by the matrix $I - \beta P^\pi$. This means that similarly to $\boldsymbol{v}$, the value function $\boldsymbol{u}(\lambda)$ is affine in λ.

Hence, following the same development in Section 4, we can modify Algorithm 2 to compute the discounted Whittle index by modifying only the initialization phase:

$$\tilde{\Delta} = \beta(P^1 - P^0) \text{ and } X^1 = \tilde{\Delta}(I - \beta P^{\pi^1})^{-1}.$$

[6] Note that the definition of $\tilde{\Delta}$ is identical to Δ except when $j = 1$, for which $\Delta_{i1} := 0$ but $\tilde{\Delta}_{i1} = \beta(P_{i1}^1 - P_{i1}^0)$.

Note that we still have $\boldsymbol{y}^1 = \boldsymbol{0}$ because $(I - \beta P^{\pi^1})^{-1}\boldsymbol{\pi}^1 = \frac{1}{1-\beta}\boldsymbol{1}$ (value function in a β-discounted Markov chain with reward equals to 1 in all states) and $\tilde{\Delta}\boldsymbol{1} = \boldsymbol{0}$. Also, if Subroutine 4 is used, Line 3 should be changed to $X^{k_0} = \tilde{\Delta}(I - \beta P^{\pi^{k_0}})^{-1}$. Last but not least, in the discounted case, the unichain property is no longer needed because the matrix $(I - \beta P^{\pi})$ is invertible for any policy π as long as $\beta < 1$ (from Perron-Frobenius' Theorem).

7.3 Gittins Index

The notion of "restless" bandit comes from the fact that even when the action "rest" is taken, the Markov chain can still change state and generate rewards. When this is not the case (*i.e.*, when $P^0 = I$ and $r^0 = 0$), a RB is no longer restless and is simply called a Markovian bandit (or a *rested* Markovian bandit if one wants to emphasize that it is not restless).

In a discounted rested bandit, the notion of Whittle index coincides[7] with the notion of Gittins index. In such a case, there is no notion of indexability: a discounted rested bandit is always indexable. Its index can be computed by Algorithm 2 without testing indexability. The best known algorithm to compute Gittins index runs in $(2/3)n^3 + O(n^2)$ [16]. When using fast multiplication, our algorithm computes Gittins index in $O(n^{2.5286})$ which makes it the first algorithm to compute Gittins index in subcubic time.

Note that when $P^1 = I$, it is possible to compute $X^1 = \tilde{\Delta}/(1-\beta)$ without having to solve a linear system which means that, when $P^1 = I$, our algorithm with the variant Algorithm 3 has the complexity $(2/3)n^3 + O(n^2)$ complexity. This shows that our cubic algorithm can compute the Gittins index in $(2/3)n^3 + O(n^2)$ by switching $(P^0, \boldsymbol{r}^0)$ and $(P^1, \boldsymbol{r}^1)$.

8 CONCLUSION

In this paper, we present an algorithm that is efficient for detecting the non-indexability and computing Whittle index of all indexable finite-state restless bandits. Our algorithm is based on the efficient application of Sherman-Morrison formula. This algorithm works for both discounted and non-discounted restless bandits, and can also be used for Gittins index computation. We present a first version of our algorithm that runs in $n^3 + o(n^3)$ arithmetic operations (or in $(2/3)n^3 + o(n^3)$ if we do not test the indexability). So, we conclude that Whittle index is not harder to compute than Gittins index. The second version of our algorithm uses fast matrix multiplication and has a complexity of $O(n^{2.5286})$. This makes it the first subcubic algorithm to compute Whittle index or Gittins index. We provide numerical simulations that show that our algorithm is very efficient in practice: it can test indexability and compute the index of a restless bandit process in less than one second for $n = 1000$ states, and in a few minutes for $n = 15000$. These numbers are provided for dense matrices. One might expect to have more efficient algorithms if the process has a sparse structure. We leave this question for future work.

[7] In fact, Whittle index was first introduced as a generalization of Gittins index to restless bandit in [47].

REFERENCES

[1] Samuli Aalto, Urtzi Ayesta, and Rhonda Righter. 2009. On the Gittins index in the M/G/1 queue. *Queueing Systems* 63, 1-4 (2009), 437.

[2] Samuli Aalto, Urtzi Ayesta, and Rhonda Righter. 2011. Properties of the Gittins index with application to optimal scheduling. *Probability in the Engineering and Informational Sciences* 25, 3 (2011), 269–288.

[3] Samuli Aalto, Pasi Lassila, and Ianire Taboada. 2019. Whittle index approach to opportunistic scheduling with partial channel information. *Performance Evaluation* 136 (2019), 102052.

[4] Nima Akbarzadeh and Aditya Mahajan. 2019. Restless bandits with controlled restarts: Indexability and computation of Whittle index. In *2019 IEEE 58th Conference on Decision and Control (CDC)*. IEEE, 7294–7300.

[5] Nima Akbarzadeh and Aditya Mahajan. 2020. Conditions for indexability of restless bandits and an algorithm to compute Whittle index. *arXiv preprint arXiv:2008.06111* (2020).

[6] Nima Akbarzadeh and Aditya Mahajan. 2020. Restless bandits: indexability and computation of Whittle index. *arXiv e-prints* (2020), arXiv–2008.

[7] Nima Akbarzadeh and Aditya Mahajan. 2021. Maintenance of a collection of machines under partial observability: Indexability and computation of Whittle index. *arXiv preprint arXiv:2104.05151* (2021).

[8] Josh Alman and Virginia Vassilevska Williams. 2021. A refined laser method and faster matrix multiplication. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 522–539.

[9] PS Ansell, Kevin D Glazebrook, José Nino-Mora, and M O'Keeffe. 2003. Whittle's index policy for a multi-class queueing system with convex holding costs. *Mathematical Methods of Operations Research* 57, 1 (2003), 21–39.

[10] Thomas W Archibald, DP Black, and Kevin D Glazebrook. 2009. Indexability and index heuristics for a simple class of inventory routing problems. *Operations research* 57, 2 (2009), 314–326.

[11] Konstantin Avrachenkov, Urtzi Ayesta, Josu Doncel, and Peter Jacko. 2013. Congestion control of TCP flows in Internet routers by means of index policy. *Computer Networks* 57, 17 (2013), 3463–3478.

[12] Konstantin Avrachenkov and Vivek S Borkar. 2020. Whittle index based Q-learning for restless bandits with average reward. *arXiv preprint arXiv:2004.14427* (2020).

[13] Konstantin Avrachenkov, Alexei Piunovskiy, and Yi Zhang. 2018. Impulsive control for G-AIMD dynamics with relaxed and hard constraints. In *2018 IEEE Conference on Decision and Control (CDC)*. IEEE, 880–887.

[14] Urtzi Ayesta, Manu K Gupta, and Ina Maria Verloop. 2021. On the computation of Whittle's index for Markovian restless bandits. *Mathematical Methods of Operations Research* 93, 1 (2021), 179–208.

[15] Vivek S Borkar and Sarath Pattathil. 2017. Whittle indexability in egalitarian processor sharing systems. *Annals of Operations Research* (2017), 1–21.

[16] Jhelum Chakravorty and Aditya Mahajan. 2014. Multi-Armed Bandits, Gittins Index, and Its Calculation. In *Methods and Applications of Statistics in Clinical Trials*, N. Balakrishnan (Ed.). John Wiley & Sons, Inc., Hoboken, NJ, USA, 416–435. https://doi.org/10.1002/9781118596333.ch24

[17] Yih Ren Chen and Michael N Katehakis. 1986. Linear programming for finite state multi-armed bandit problems. *Mathematics of Operations Research* 11, 1 (1986), 180–183.

[18] Don Coppersmith and Shmuel Winograd. 1987. Matrix multiplication via arithmetic progressions. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. 1–6.

[19] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to algorithms*. MIT press.

[20] Jing Fu, Yoni Nazarathy, Sarat Moka, and Peter G Taylor. 2019. Towards q-learning the whittle index for restless bandits. In *2019 Australian & New Zealand Control Conference (ANZCC)*. IEEE, 249–254.

[21] François Le Gall and Florent Urrutia. 2018. Improved rectangular matrix multiplication using powers of the Coppersmith-Winograd tensor. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 1029–1046.

[22] Nicolas Gast, Bruno Gaujal, and Chen Yan. 2020. Exponential Convergence Rate for the Asymptotic Optimality of Whittle Index Policy. *arXiv preprint arXiv:2012.09064* (2020).

[23] J. C. Gittins. 1979. Bandit Processes and Dynamic Allocation Indices. *Journal of the Royal Statistical Society: Series B (Methodological)* 41, 2 (Jan. 1979), 148–164. https://doi.org/10.1111/j.2517-6161.1979.tb01068.x

[24] KD Glazebrook and HM Mitchell. 2002. An index policy for a stochastic scheduling model with improving/deteriorating jobs. *Naval Research Logistics (NRL)* 49, 7 (2002), 706–721.

[25] Kevin D Glazebrook, Christopher Kirkbride, and Jamal Ouenniche. 2009. Index policies for the admission control and routing of impatient customers to heterogeneous service stations. *Operations Research* 57, 4 (2009), 975–989.

[26] Kevin D Glazebrook, Diego Ruiz-Hernandez, and Christopher Kirkbride. 2006. Some indexable families of restless bandit problems. *Advances in Applied Probability* 38, 3 (2006), 643–672.

[27] Jianyu Huang et al. 2018. *Practical fast matrix multiplication algorithms*. Ph.D. Dissertation.

[28] Jianyu Huang, Tyler M Smith, Greg M Henry, and Robert A Van De Geijn. 2016. Strassen's algorithm reloaded. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 690–701.

[29] Xiaohan Huang and Victor Y Pan. 1998. Fast rectangular matrix multiplication and applications. *Journal of complexity* 14, 2 (1998), 257–299.

[30] Michael N Katehakis and Arthur F Veinott Jr. 1987. The multi-armed bandit problem: decomposition and computation. *Mathematics of Operations Research* 12, 2 (1987), 262–268.

[31] Maialen Larrañaga, Urtzi Ayesta, and Ina Maria Verloop. 2015. Asymptotically optimal index policies for an abandonment queue with convex holding cost. *Queueing systems* 81, 2 (2015), 99–169.

[32] Keqin Liu and Qing Zhao. 2010. Indexability of restless bandit problems and optimality of whittle index for dynamic multichannel access. *IEEE Transactions on Information Theory* 56, 11 (2010), 5547–5567.

[33] Christopher Lott and Demosthenis Teneketzis. 2000. On the optimality of an index rule in multichannel allocation for single-hop mobile networks with multiple service classes. *Probability in the Engineering and Informational Sciences* 14, 3 (2000), 259–297.

[34] Khaled Nakhleh, Santosh Ganji, Ping-Chun Hsieh, I-Hong Hou, and Srinivas Shakkottai. 2022. NeurWIN: Neural Whittle Index Network For Restless Bandits Via Deep RL. arXiv:2110.02128 [cs.LG]

[35] José Niño-Mora. 2007. A (2/3) n^3 Fast-Pivoting Algorithm for the Gittins Index and Optimal Stopping of a Markov Chain. *INFORMS Journal on Computing* 19, 4 (Nov. 2007), 596–606. https://doi.org/10.1287/ijoc.1060.0206

[36] José Niño-Mora. 2007. Dynamic priority allocation via restless bandit marginal productivity indices. *Top* 15, 2 (2007), 161–198.

[37] José Niño-Mora. 2014. A dynamic page-refresh index policy for web crawlers. In *International Conference on Analytical and Stochastic Modeling Techniques and Applications*. Springer, 46–60.

[38] José Niño-Mora. 2020. A fast-pivoting algorithm for Whittle's restless bandit index. *Mathematics* 8, 12 (2020), 2226.

[39] V Ya Pan. 1972. On schemes for the computation of products and inverses of matrices. *Russian Math. Surveys* 27, 5 (1972), 249–250.

[40] Christos H Papadimitriou and John N Tsitsiklis. 1994. The complexity of optimal queueing network control. In *Proceedings of IEEE 9th Annual Conference on Structure in Complexity Theory*. IEEE, 318–322.

[41] Martin L. Puterman. 1994. *Markov Decision Processes: Discrete Stochastic Dynamic Programming* (1st ed.). John Wiley & Sons, Inc., USA.

[42] Ziv Scully, Mor Harchol-Balter, and Alan Scheller-Wolf. 2018. SOAP: One clean analysis of all age-based scheduling policies. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 2, 1 (2018), 1–30.

[43] Isaac M Sonin. 2008. A generalized Gittins index for a Markov chain and its recursive calculation. *Statistics & Probability Letters* 78, 12 (2008), 1526–1533.

[44] Volker Strassen. 1969. Gaussian elimination is not optimal. *Numerische mathematik* 13, 4 (1969), 354–356.

[45] Sofía S Villar, Jack Bowden, and James Wason. 2015. Multi-armed bandit models for the optimal design of clinical trials: benefits and challenges. *Statistical science: a review journal of the Institute of Mathematical Statistics* 30, 2 (2015), 199.

[46] Richard R Weber and Gideon Weiss. 1990. On an index policy for restless bandits. *Journal of applied probability* (1990), 637–648.

[47] Peter Whittle. 1988. Restless bandits: Activity allocation in a changing world. *Journal of applied probability* 25, A (1988), 287–298.

[48] Max A Woodbury. 1950. *Inverting modified matrices.* Statistical Research Group.

A TECHNICAL LEMMAS

A.1 Minimal Optimal policy

LEMMA A.1. *Let λ be a fixed penalty and consider that there exist two distinct optimal policies $\pi \neq \pi'$. Then:*

(1) The policy $\pi \cap \pi'$ is also optimal for this penalty.
(2) If $\pi \supset \pi'$, then for $i \in \pi \supset \pi'$, $\pi \setminus \{i\}$ is also optimal for this penalty.

Proof. By Section 8.4 of [41], and since the RB is unichain, there exists a bias vector $h^*(\lambda)$ such that π is an optimal policy if and only if for all state i:

$$r_i^1 - \lambda + \sum_{j=2}^{n} P_{ij}^1 h_j^*(\lambda) > r_i^0 + \sum_{j=2}^{n} P_{ij}^0 h_j^*(\lambda) \qquad \text{implies } i \in \pi; \tag{20}$$

$$r_i^1 - \lambda + \sum_{j=2}^{n} P_{ij}^1 h_j^*(\lambda) < r_i^0 + \sum_{j=2}^{n} P_{ij}^0 h_j^*(\lambda) \qquad \text{implies } i \notin \pi. \tag{21}$$

It should be clear that if both π and π' satisfy (20)–(21), then so does $\pi \cap \pi'$ which implies that it is also an optimal policy.

Note that in particular, if $\pi \supsetneq \pi'$ are two optimal policies, then let $i \in \pi \setminus \pi'$. This state must be such that $r_i^1 - \lambda + \sum_{j=2}^{n} P_{ij}^1 h_j^*(\lambda) = r_i^0 + \sum_{j=2}^{n} P_{ij}^0 h_j^*(\lambda)$ which implies that $\pi \setminus \{i\}$ satisfies (20)–(21) and is therefore also an optimal policy. □

A.2 Unichain property

Lemma A.2. *Given a MDP $\langle [n], \{0,1\}, r, P \rangle$, let P^π be the transition matrix under policy π. If the MDP is* unichain*, the matrix*

$$A^\pi = \begin{bmatrix} 1 & -P_{12}^\pi & \dots & -P_{1n}^\pi \\ 1 & 1 - P_{22}^\pi & \dots & -P_{2n}^\pi \\ \vdots & & & \\ 1 & -P_{n2}^\pi & \dots & 1 - P_{nn}^\pi \end{bmatrix}$$

is invertible for any policy π.

Proof. A^π is not invertible, if there exists a vector $\boldsymbol{u} \neq \boldsymbol{0}$ such that $\boldsymbol{u}^T A^\pi = \boldsymbol{0}$. We proof that such $\boldsymbol{u}$ does not exist when the MDP is unichain. Let $\boldsymbol{u} \in \mathbb{R}^n$ be an arbitrary vector such that $\boldsymbol{u}^T A^\pi = \boldsymbol{0}$. Then, we have

$$\begin{cases} \sum_{i=1}^{n} u_i & = 0 \\ u_i - \sum_{j=1}^{n} u_j P_{ji}^\pi & = 0, \text{ for } 2 \le i \le n \end{cases}$$

Then,

$$\begin{aligned} \sum_{i=1}^{n} u_i &= u_1 + \sum_{i=2}^{n} u_i \\ 0 &= u_1 + \sum_{i=2}^{n} \sum_{j=1}^{n} u_j P_{ji}^\pi = u_1 + \sum_{j=1}^{n} u_j \sum_{i=2}^{n} P_{ji} = u_1 + \sum_{j=1}^{n} u_j (1 - P_{j1}^\pi) \\ 0 &= u_1 - \sum_{j=1}^{n} u_j P_{j1}^\pi \end{aligned}$$

Then, we get

$$\begin{cases} \sum_{i=1}^{n} u_i & = 0 \\ \boldsymbol{u}^T P^\pi & = \boldsymbol{u}^T \end{cases}$$

In unichain MDP, for any policy π, the only vectors $\boldsymbol{u}$ verifying $\boldsymbol{u}^T P^\pi = \boldsymbol{u}^T$ are multiples of the stationary distribution under policy π [41]. Therefore $\sum_{i=1}^{n} u_i = 0$ implies $\boldsymbol{u} = 0$. □

A.3 Proof of Lemma 3.2

Proof. Let $\boldsymbol{v}^\pi(\lambda)$ be as in Equation (5). By definition, the matrices A^π and $A^{\pi\setminus\{i\}}$ differ only in row i, that is, for $i \in \pi$

$$
\begin{aligned}
(A^\pi)_i &= [1 \quad -P^1_{i2} \quad \dots \quad -P^1_{ii-1} \quad 1-P^1_{ii} \quad -P^1_{ii+1} \quad \dots \quad -P^1_{in}] \\
\text{and } (A^{\pi\setminus\{i\}})_i &= [1 \quad -P^0_{i2} \quad \dots \quad -P^0_{ii-1} \quad 1-P^0_{ii} \quad -P^0_{ii+1} \quad \dots \quad -P^0_{in}].
\end{aligned}
$$

This implies that (3) with $\pi\setminus\{i\}$ holds for all $j \neq i$: $(A^\pi v^\pi(\lambda))_j = (A^{\pi\setminus\{i\}} v^\pi(\lambda))_j = r_j^{\pi_j} - \lambda\pi_j$. Moreover, if λ is such that (6) holds, then

$$
r_i^1 - \lambda + \sum_{j=2}^n P^1_{ij} v^\pi_j(\lambda) = r_i^0 + \sum_{j=2}^n P^0_{ij} v^\pi_j(\lambda).
$$

So, for any $i \in \pi$,

$$
g^\pi(\lambda) + v^\pi_i(\lambda) = r_i^1 - \lambda + \sum_{j=2}^n P^1_{ij} v^\pi_j(\lambda) = r_i^0 + \sum_{j=2}^n P^0_{ij} v^\pi_j(\lambda)
$$

$$
g^\pi(\lambda) + v^\pi_i(\lambda) - \sum_{j=2}^n P^0_{ij} v^\pi_j(\lambda) = r_i^0 \text{ or } (A^{\pi\setminus\{i\}} v^\pi(\lambda))_i = r_i^0.
$$

This implies that $\boldsymbol{v}^\pi(\lambda)$ is a solution of $A^{\pi\setminus\{i\}}\boldsymbol{v}^\pi(\lambda) = \boldsymbol{r}^{\pi\setminus\{i\}} - \lambda(\boldsymbol{\pi} - \boldsymbol{e}_i)$, which implies that $v^\pi(\lambda) = v^{\pi\setminus\{i\}}(\lambda)$. Note that $\boldsymbol{e}_i$ is the column vector where all entries are equal to 0 except for the i-th, equal to 1. □

B IMPLEMENTATIONS

B.1 Arithmetic complexity of Algorithm 3 and memory usage

Recall that in Algorithm 3, we compute the values X^ℓ_{ij} by doing the update (for all iteration k, for all $\ell = 1$ to k and for all $i \in [n]$ or all $i \in \pi^{\ell+1}$ if we do not test indexability):

$$
X^{\ell+1}_{i\sigma^k} := X^\ell_{i\sigma^k} - \frac{X^\ell_{i\sigma^\ell}}{1 + X^\ell_{\sigma^\ell\sigma^\ell}} X^\ell_{\sigma^\ell\sigma^k} \tag{22}
$$

If we test indexability, there are $\sum_{k=1}^n kn = n^3/2 + O(n^2)$ such updates. If we do not test indexability, there are $\sum_{k=1}^n \sum_{\ell=1}^k (n-\ell) = n^3/3 + O(n^2)$ such updates. Below, we show each update of Equation (22) can be done in two arithmetic operations (one addition and one multiplication), which leads to the complexity of $n^3 + O(n^2)$ (or $(2/3)n^3 + O(n^2)$) arithmetic operations for the computation of all the needed X^k_{ij}. We also show how to reduce the memory size to $O(n^2)$.

Let $W_{ik} = X^k_{i\sigma^k}/(1 + X^k_{\sigma^k\sigma^k})$ and $V^k_{i\ell} = X^\ell_{i\sigma^k}$. Using this, Equation (22) can be rewritten as:

$$
V^k_{i\ell+1} = V^k_{i\ell} - W_{i\ell} V^k_{\sigma^\ell\ell}, \tag{23}
$$

with $W_{ik} = V^k_{ik}/(1 + V^k_{\sigma^k k})$.

This results in the following loop at iteration k:

- Initialize V^k_{i1} from $X^1_{i\sigma^k}$.
- For all $\ell \in \{1 \dots k-1\}$, and all $i \in [n]$ (or $i \in \pi^{\ell+1}$), apply (23).
- Compute $W_{ik} = V^k_{ik}/(1 + V^k_{\sigma^k k})$
- For $\ell = k$, and all $i \in [n]$ (or $i \in \pi^{\ell+1}$), apply (23).

Note that the value of V^{k-1} is not necessary for iteration k (only the values of $W_{i\ell}$ are needed). This shows that the algorithm can be implemented with a memory $O(n^2)$. Moreover, each update of (22) can be implemented by 2 arithmetic operations (one multiplication and one subtraction).

B.2 Speedup when not checking the indexability: First found Go last

When the indexability is not tested, the update (23) is computed for all $i \in \pi^{\ell+1}$. This creates inefficiencies (due to inefficient cache usage) because the elements $V^k_{i\ell+1}$ are not accessed sequentially.

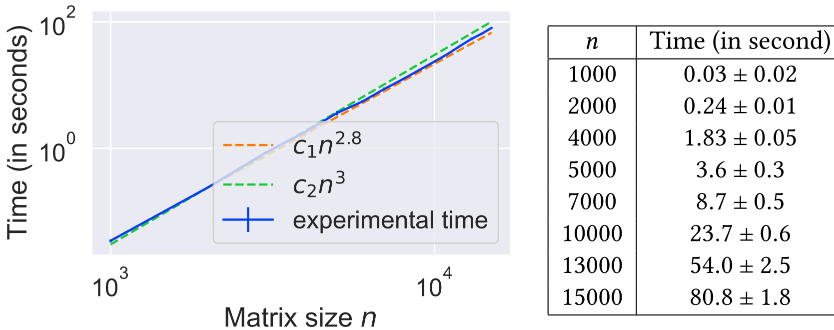
To speedup the memory accesses, our solution is to sort the items during the execution of the algorithm. At iteration k, the algorithm computes σ^k. When this is done, our implementation switches all quantities in positions σ^k and $n-k+1$. These quantities are δ, y, z, W and X. For instance, once σ^1 is found, we know that the state at position n is state n and we do the following switches:

$$\begin{aligned} \delta_{\sigma^1}, \delta_n &\to \delta_n, \delta_{\sigma^1} \\ y^1_{\sigma^1}, y^1_n &\to y^1_n, y^1_{\sigma^1} \\ z^1_{\sigma^1}, z^1_n &\to z^1_n, z^1_{\sigma^1} \\ W_{\sigma^1}, W_n &\to W_n, W_{\sigma^1} \\ \text{and } X_{\sigma^1}, X_n &\to X_n, X_{\sigma^1}. \end{aligned}$$

To do so, we need an array to store all states such that at iteration k, the first $n-k$ states of the array are the active states. We will need to track the position of each state in such array.

C ANALYSIS OF THE EXPERIMENTAL TIME TO SOLVE A LINEAR SYSTEM

In this section, we report in Figure 6 the time taken by the default implementation to solve a linear system of the form $AX = B$ where A and B are two square matrices. To obtain this figure, we generated random (full) matrices where each entry is between 0 and 1 and use the function `scipy.linalg.solve` from the library `scipy`. The reported numbers suggest that the complexity of the solver is closer to $O(n^{2.8})$ than to $O(n^3)$, although we agree that the difference between the $O(n^{2.8})$ and the $O(n^3)$ curves is small. Note that this is in accordance with the papers [27, 28] that claim that the fastest implementations of matrix multiplication and inversion are based on Strassen's algorithm and should therefore be in $O(n^{2.8})$.



n	Time (in second)
1000	0.03 ± 0.02
2000	0.24 ± 0.01
4000	1.83 ± 0.05
5000	3.6 ± 0.3
7000	8.7 ± 0.5
10000	23.7 ± 0.6
13000	54.0 ± 2.5
15000	80.8 ± 1.8

Fig. 6. Time taken of the default implementations `scipy.linalg.solve` of scipy to solve a linear system $AX = B$ where A and B are two square $n \times n$ matrices.

D DETAILED COMPARISON WITH [5] AND [38]

In this section, we compare our algorithm with two main related works for finite-state restless bandits problem.

D.1 Comparison with [5]

The paper presents an algorithm that computes Whittle indices in $O(n^3)$ (no explicit constant before n^3 is given) for all indexable problems. Despite following a different approach, our algorithm for computing Whittle index can be viewed as a refinement of this work. Let us recall once again that our approach also allows one to check the indexability of general restless bandits.

In the following, we show how we can refine the work of [5] to obtain an algorithm that is exactly the same as ours. Let D^π and N^π be two vectors defined as in [5],

$$D^\pi = (1-\beta)(I-\beta P^\pi)^{-1}\boldsymbol{r}^\pi, \quad \text{and} \quad N^\pi = (1-\beta)(I-\beta P^\pi)^{-1}\boldsymbol{\pi}.$$

Then, we have $D^\pi - \lambda N^\pi = (1-\beta)\boldsymbol{u}^\pi(\lambda)$ where $\boldsymbol{u}^\pi(\lambda)$ is defined as in (17). In our proposition, at each iteration k, we compute μ_i^k by Line 7. Instead, it is defined in [5] by two steps:

(1) for all state $j \in [n]$ such that $N_j^{\pi^k \setminus \{i\}} \neq N_j^{\pi^k}$, compute $\mu_{ij}^k = \dfrac{D_j^{\pi^k \setminus \{i\}} - D_j^{\pi^k}}{N_j^{\pi^k \setminus \{i\}} - N_j^{\pi^k}}$

(2) compute $\mu_i^k = \operatorname*{arg\,min}_{j \in [n] : N_j^{\pi^k \setminus \{i\}} \neq N_j^{\pi^k}} \mu_{ij}^k$.

From [5, Theorem 2], in an indexable problem, for state σ^k, there exists a state $j \in [n]$ such that $N_j^{\pi^k \setminus \{\sigma^k\}} \neq N_j^{\pi^k}$. Now, suppose that for any active state $i \in \pi^k$, there exists $j \in [n]$ such that $N_j^{\pi^k \setminus \{i\}} \neq N_j^{\pi^k}$. Using Sherman-Morrison formula, we have[8]

$$D^{\pi^k \setminus \{i\}} - D^{\pi^k} = -\frac{(1-\beta)\delta_i + \tilde{\Delta}_i D^{\pi^k}}{1 + \tilde{\Delta}_i [(I-\beta P^{\pi^k})^{-1}]_{:i}} [(I-\beta P^{\pi^k})^{-1}]_{:i},$$

$$\text{and} \quad N^{\pi^k \setminus \{i\}} - N^{\pi^k} = -\frac{(1-\beta) + \tilde{\Delta}_i N^{\pi^k}}{1 + \tilde{\Delta}_i [(I-\beta P^{\pi^k})^{-1}]_{:i}} [(I-\beta P^{\pi^k})^{-1}]_{:i}.$$

Then, for any $j \in [n]$ such that $N_j^{\pi^k \setminus \{i\}} \neq N_j^{\pi^k}$, $\mu_{ij}^k = \dfrac{(1-\beta)\delta_i + \tilde{\Delta}_i D^{\pi^k}}{(1-\beta) + \tilde{\Delta}_i N^{\pi^k}}$ which does not depend on j. Then, we simply have $\mu_i^k = \dfrac{(1-\beta)\delta_i + \tilde{\Delta}_i D^{\pi^k}}{(1-\beta) + \tilde{\Delta}_i N^{\pi^k}}$. Also, we have

$$\tilde{\Delta}_i N^{\pi^k} = (1-\beta)\tilde{\Delta}_i (I-\beta P^{\pi^k})^{-1}\boldsymbol{\pi}^k = -(1-\beta) y_i^k \quad \text{and}$$
$$\tilde{\Delta}_i D^{\pi^k} = (1-\beta)\tilde{\Delta}_i \boldsymbol{u}^{\pi^k}(\mu_{\min}^{k-1}) + \mu_{\min}^{k-1}\tilde{\Delta}_i N^{\pi^k} = (1-\beta) z_i^{k-1} - (1-\beta)\mu_{\min}^{k-1} y_i^k.$$

So, replacing these terms in μ_i^k, we get the formula given at Line 7.

Note that the algorithm of [5] was only developed for the discounted case. Our approach for the time-average reward case is different because we use the average gain and bias instead of directly the expected discounted total reward D^{π^k} and total number of activations N^{π^k} under policy π^k. Moreover, our code is also optimized to avoid unnecessary computation and to reduce memory usage. Finally, the way we do the update of our matrix X makes it possible to obtain a subcubic algorithm whereas their approach does not (see also below).

[8] the expression of $D^{\pi \setminus \{i\}}$ and $N^{\pi \setminus \{i\}}$ given by Equation (18) in [5] are erroneous.

D.2 Comparison with algorithm of [38]

The algorithm [38] had the best complexity up to date, there is a square matrix $\boldsymbol{A}$ that plays a similar role as the square matrix X in our proposed algorithm. The most costly operations in the algorithm of [38] is to update their matrix $\boldsymbol{A}$ at each iteration and it is done by Equation (15) that we recall here (using the same notation $\boldsymbol{A}$ as in their paper):

$$\text{for } i, j \in \pi^k,\ \boldsymbol{A}^k_{ij} = \boldsymbol{A}^{k-1}_{ij} - \frac{\boldsymbol{A}^{k-1}_{i\sigma^k}}{\boldsymbol{A}^{k-1}_{\sigma^k\sigma^k}} \boldsymbol{A}^{k-1}_{\sigma^k j}. \tag{24}$$

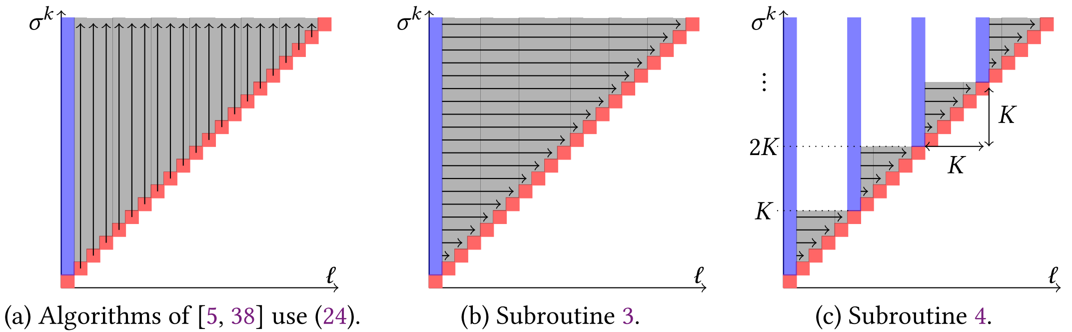
This incurs a total complexity of $(2/3)n^3 + O(n^2)$ arithmetic operations. As mentioned in Section 4.3, if we updated X^k as given by (15), our algorithm would also have a $(2/3)n^3 + O(n^2)$ complexity but this version of the update cannot be optimized by using fast matrix multiplication.

D.3 The approach of [5, 38] cannot be directly transformed into a subcubic algorithm

In addition to all the previously cited differences, one of the major contribution of our algorithm with respect to [5, 38] is that the most advanced version of our algorithm runs in a subcubic time. The approach[9] used in [5, 38] is to update the full matrix X^k at iteration k, by using (24). This idea is represented in Figure 7(a): for a given ℓ, their algorithm compute $X^\ell_{:\sigma^k}$ for all k (i.e., the full vertical lines represented by arrows). Our first Subroutine 3 uses an horizontal approach based on (16), which we recall here:

$$X^{\ell+1}_{i\sigma^k} = X^\ell_{i\sigma^k} - \frac{X^\ell_{i\sigma^\ell} X^\ell_{\sigma^\ell\sigma^k}}{1 + X^\ell_{\sigma^\ell\sigma^\ell}}.$$

At iteration k, we use $X^1_{:\sigma^k}$ to compute all values of $X^\ell_{:\sigma^k}$ up to $\ell = k + 1$. This is represented in Figure 7(b). Our approach can be used to obtain the subcubic algorithm illustrated in Figure 7(c) by using subcubic algorithms for multiplication.



(a) Algorithms of [5, 38] use (24). (b) Subroutine 3. (c) Subroutine 4.

Fig. 7. Comparison of the implementation of (24) used in [6, 38] with Subroutine 3 and Subroutine 4.

This leads to the next fundamental question: why should the computation of Whittle index be harder than matrix inversion (or multiplication)? To us, the main difference is that when computing Whittle indices, the permutation σ is not known a priori but is discovered as the algorithm progresses: σ^k is only known at iteration k. Hence, while all terms of the matrices X^k_{ij} are not needed, it is difficult to know a priori which ones are needed and which ones are not. Hence, a simple divide and conquer algorithm cannot be used. This is why when recomputing X^k

[9] Equation (18) of [5], which is central to their algorithm is the same as the above equation (24).

in Subroutine 4, we recompute the whole matrix (the vertical blue line) and not just the part that will be used to compute the gray zone: we do not know *a priori* what part of X_{ij}^k will be useful or not.