# Lazy and Fast Greedy MAP Inference for Determinantal Point Process

**Shinichi Hemmi**
The University of Tokyo, Tokyo, Japan
hemmi.shinichi@gmail.com

**Taihei Oki**
The University of Tokyo, Tokyo, Japan
oki@mist.i.u-tokyo.ac.jp

**Shinsaku Sakaue**
The University of Tokyo, Tokyo, Japan
sakaue@mist.i.u-tokyo.ac.jp

**Kaito Fujii**
National Institute of Informatics, Tokyo, Japan
fujiik@nii.ac.jp

**Satoru Iwata**
The University of Tokyo, Tokyo, Japan
ICReDD, Hokkaido University, Sapporo, Japan
iwata@mist.i.u-tokyo.ac.jp

## Abstract

The maximum a posteriori (MAP) inference for determinantal point processes (DPPs) is crucial for selecting diverse items in many machine learning applications. Although DPP MAP inference is NP-hard, the greedy algorithm often finds high-quality solutions, and many researchers have studied its efficient implementation. One classical and practical method is the lazy greedy algorithm, which is applicable to general submodular function maximization, while a recent fast greedy algorithm based on the Cholesky factorization is more efficient for DPP MAP inference. This paper presents how to combine the ideas of "lazy" and "fast", which have been considered incompatible in the literature. Our lazy and fast greedy algorithm achieves almost the same time complexity as the current best one and runs faster in practice. The idea of "lazy + fast" is extendable to other greedy-type algorithms. We also give a fast version of the double greedy algorithm for unconstrained DPP MAP inference. Experiments validate the effectiveness of our acceleration ideas.

## 1 Introduction

Determinantal point processes (DPPs) offer a popular diversification model in machine learning. Macchi [34] first used DPPs to represent repulsion in quantum physics, and later, DPPs have been used in various scenarios such as recommendation systems [13], document summarization [21, 31], and diverse molecule selection [39]. An important problem in the DPP applications is the *maximum a posteriori* (MAP) inference, which asks to find an item subset with the highest probability. Intuitively, if each item is associated with a vector whose length and direction represent its importance and feature, respectively, then the aim of DPP MAP inference is to select items whose vectors form the largest volume parallelotope, thus selecting important and diverse items.

In DPPs, exact MAP inference is NP-hard [27]. Fortunately, however, the standard greedy algorithm (GREEDY) for submodular function maximization [41] enjoys a $(1 - 1/\mathrm{e})$-approximation guarantee in terms of the log-determinant function value under the monotonicity assumption, and it often finds high-quality solutions in practice. A naive implementation of GREEDY, however, incurs too much computation cost for large instances since evaluating the determinant of a $k \times k$ matrix takes $\mathrm{O}(k^\omega)$

36th Conference on Neural Information Processing Systems (NeurIPS 2022).

time, where $\omega \in [2, 3]$ is the matrix-multiplication exponent (usually $\omega = 3$). To overcome this issue, researchers have studied techniques for implementing efficient greedy algorithms. One classical and powerful method is the so-called *lazy* greedy algorithm (LAZYGREEDY) [37], which avoids the redundant computation of function values by making good use of the submodularity. LAZYGREEDY is applicable to submodular function maximization, and it empirically runs much faster than naive GREEDY, although the worst-case running time is not improved. Chen et al. [13] proposed another notable Cholesky-factorization-based method, called the *fast* greedy algorithm (FASTGREEDY). Their algorithm is specialized for DPP MAP inference and provides the fastest $\mathrm{O}(knd)$-time implementation of GREEDY for selecting $k$ out of $n$ items represented by $d$-dimensional vectors. Chen et al. [13] also experimentally showed that FASTGREEDY can run faster than LAZYGREEDY.

Since the study of [13], although a pre-processing method for customized DPP MAP inference [23] and fast parallel algorithms for submodular function maximization [3, 4, 18, 8, 17, 14, 29] have been studied, no progress has been made that directly accelerates GREEDY for DPP MAP inference. Since real-world dataset sizes have been growing, a further speed-up of GREEDY is eagerly awaited.

**Our main contribution** is to combine the two ideas, "lazy" and "fast", to develop an even faster implementation of GREEDY for DPP MAP inference, which we call LAZYFASTGREEDY. In the literature, the two ideas have been thought to be incompatible; in fact, experiments in [13] compared FASTGREEDY with LAZYGREEDY without considering their combination. The core idea of "lazy + fast" is widely applicable to other greedy-type algorithms. Below is a summary of our results.

1. We present LAZYFASTGREEDY for cardinality-constrained DPP MAP inference. It takes $\mathrm{O}(kn(d+\log n))$ time even in the worst case and is faster than FASTGREEDY in practice. We also extend the idea of "lazy + fast" to other greedy-type algorithms: RANDOMGREEDY [10], STOCHASTICGREEDY [38, 43], and INTERLACEGREEDY [28].
2. We present a "fast" version of DOUBLEGREEDY [11] for unconstrained DPP MAP inference by extending the idea of [13] with Jacobi's complementary minor formula.
3. Experiments on synthetic and real-world datasets validate the empirical effectiveness of our acceleration techniques for the greedy-type algorithms. In particular, our LAZYFASTGREEDY runs up to about 17 times faster than FASTGREEDY in real-world settings.

Accelerating the greedy variants [10, 38, 28, 43], which enjoy approximation guarantees for non-monotone submodular function maximization, is worthwhile since the log-determinant function is non-monotone in general. Also, note that both cardinality-constrained and unconstrained settings are important in DPP MAP inference and require substantially different technical ideas. Therefore, we study the fast version of [11] separately from the algorithms for the cardinality-constrained setting. In short, we accelerate various important greedy-type algorithms for DPP MAP inference.

### 1.1 Related work

The greedy algorithm (GREEDY) is a popular approach to DPP MAP inference [31]. Han et al. [24] gave a fast but inexact implementation of GREEDY. Later, Chen et al. [13] gave an exact implementation of GREEDY with the same time complexity as that of [24]. Han and Gillenwater [23] studied special cases where kernel matrices of DPPs are generated via customization (or re-weighting) of fixed feature vectors and developed a pre-processing method for accelerating GREEDY. Note that this pre-processing usually takes longer than the algorithm of [13] (see [23, Section 4]), while their algorithm after pre-processing is much faster. In summary, the algorithm of Chen et al. [13] has been the fastest greedy-style algorithm for general DPPs.

A continuous-relaxation-based $1/4$-approximation algorithm for general down-closed constraints was also studied [21]. Our implementation of INTERLACEGREEDY [28] yields a faster $1/4$-approximation algorithm for a special case with a cardinality constraint. Approximation algorithms and inapproximability results of DPP MAP inference (without log) have also been extensively studied [15, 35, 36, 6, 42]. Sampling is another important research subject in DPPs and has been widely studied [2, 33, 16, 22, 1, 12, 32]. Also, we leave extension of our algorithms to non-symmetric DPPs [20], which have recently gained increasing attention, for future work.

Since the log-determinant function is known to be submodular [19], DPP MAP inference has a close connection to submodular function maximization [41]. Besides LAZYGREEDY, STOCHASTICGREEDY [38, 43] is a popular fast variant of GREEDY, which we will discuss later. A recent line of

work [3, 4, 18, 8, 17, 14, 29] has studied *adaptive* algorithms for submodular function maximization, where we are allowed to execute polynomially many queries in parallel to reduce the number of sequential rounds. Those studies consider oracle models of submodular functions, whereas we focus on the log-determinant functions and develop fast algorithms without such parallelization.

## 2 Background

Let $[n]$ denote the set $\{1, 2, \dots, n\}$ for any $n \in \mathbb{N}$. For any $S \subseteq [n]$, $\overline{S}$ denotes its complement $[n] \setminus S$. We use $\mathbf{0}$ and $O$ as all-zero vectors and matrices, respectively, and $\mathbf{1}$ as all-ones vectors (their sizes will be clear from the context). Let $\langle \cdot, \cdot \rangle$ denote the inner product. For a matrix $L \in \mathbb{R}^{n \times n}$ and subsets $S, T \subseteq [n]$, $L[S, T]$ is the submatrix of $L$ indexed by $S$ in rows and $T$ in columns. For brevity, we write $L_{i,j} = L[\{i\}, \{j\}], L[S] = L[S, S], L[S, i] = L[S, \{i\}]$, and $L[i, S] = L[\{i\}, S]$ for any $i, j \in [n]$ and $S \subseteq [n]$. The determinant of $L$ is denoted by $\det L$. Set $\det L[\emptyset] = 1$ by convention. For any $M \in \mathbb{R}^{n \times n}$, we suppose that $M^\top M$ is computed in $\mathrm{O}(n^\omega)$ time, which implies that we can compute $\det M$ and $M^{-1}$ (if non-singular) in $\mathrm{O}(n^\omega)$ time (see, e.g., [7, Chapter 2]).

**DPP MAP inference.** Let $L \in \mathbb{R}^{n \times n}$ be a positive semi-definite matrix. A probability measure $\mathcal{P}$ on $2^{[n]}$ is called a *determinantal point process* (DPP) with a kernel matrix $L$ if $\mathcal{P}[X = S] \propto \det L[S]$ holds for all $S \subseteq [n]$.[1] MAP inference for DPPs is the problem of finding a subset $S \subseteq [n]$ with the largest $\det L[S]$ value. We suppose that each $i \in [n]$ is associated with a vector $\boldsymbol{\phi}_i \in \mathbb{R}^d$ and that a kernel matrix $L \in \mathbb{R}^{n \times n}$ is given as $L = B^\top B$, where $B = [\boldsymbol{\phi}_1, \boldsymbol{\phi}_2, \dots, \boldsymbol{\phi}_n] \in \mathbb{R}^{d \times n}$, i.e., $L_{i,j} = \langle \boldsymbol{\phi}_i, \boldsymbol{\phi}_j \rangle$ for $i, j \in [n]$. Note that $\sqrt{\det L[S]}$ represents the volume of the parallelotope spanned by $\{\boldsymbol{\phi}_i \mid i \in S\}$. Hence, if the length and direction of $\boldsymbol{\phi}_i$ indicate $i$'s importance and feature, respectively, the larger value of $\det L[S]$ implies that $S$ contains more important and diverse items. In many situations, we want to select a limited number of items; let $k \in \mathbb{N}$ denote the upper bound. Therefore, MAP inference for DPPs with a cardinality constraint, or $k$-DPP [30], is often considered. Note that since $\operatorname{rank} L \leq \min\{n, d\}$, we can assume $k \leq \min\{n, d\}$ without loss of generality.

**Submodular function maximization.** For a set function $f \colon 2^{[n]} \to \mathbb{R} \cup \{-\infty\}$, the *marginal gain* of $i \in [n]$ with respect to $S \subseteq [n]$ is defined by $f_i(S) = f(S \cup \{i\}) - f(S)$. A set function $f \colon 2^{[n]} \to \mathbb{R} \cup \{-\infty\}$ is called *monotone* if $f_i(S) \geq 0$ for every $S \subseteq [n]$ and $i \in \overline{S}$. It is called *submodular* if it has the *diminishing returns property*: $f_i(S) \geq f_i(T)$ for every $S \subseteq T \subseteq [n]$ and $i \in \overline{T}$. Since $f(S) = \log \det L[S]$ is submodular, DPP MAP inference can be written as submodular function maximization: $\max_{S \in \mathcal{X}} f(S)$, where $\mathcal{X} \subseteq 2^{[n]}$ is a family of feasible subsets. This paper mostly considers the cardinality-constrained setting, i.e., $\mathcal{X} = \{S \subseteq [n] \mid |S| \leq k\}$ for given $k \in \mathbb{N}$; in Section 4, we study the unconstrained setting, i.e., $\mathcal{X} = 2^{[n]}$. The log-determinant function $f$ is monotone if the smallest eigenvalue of $L$ is at least 1 (see, e.g., [44]), but this is not always the case.

It is well-known that the greedy algorithm (GREEDY) enjoys a $(1 - 1/\mathrm{e})$-approximation guarantee for cardinality-constrained monotone submodular function maximization with $f(\emptyset) \geq 0$ [41]. This approximation ratio is optimal under the evaluation oracle model [40]. GREEDY works as follows: setting $S^{(0)} = \emptyset$, in each $t$th step ($t = 1, \dots, k$), choose $j_t \in \arg\max\{ f_i(S^{(t-1)}) \mid i \in \overline{S^{(t-1)}} \}$ and put $S^{(t)} = S^{(t-1)} \cup \{j_t\}$. We call a solution obtained in this way a *greedy solution*.

Besides GREEDY, many algorithms [11, 10, 28, 43] achieve constant-factor approximations for cardinality-constrained/unconstrained submodular function maximization. These results motivate us to apply submodular-function-maximization algorithms to DPP MAP inference, although constant-factor approximations of the log-determinant value do not imply those of the determinant value.

### 2.1 Lazy greedy algorithm for submodular function maximization

LAZYGREEDY [37] is an efficient implementation of GREEDY for submodular function maximization. As explained above, GREEDY finds $j_t$ by computing marginal gains $f_i(S^{(t-1)})$ for all $i \in \overline{S^{(t-1)}}$. LAZYGREEDY attempts to find $j_t$ more efficiently by keeping an upper bound $\rho_i$ on $f_i(S^{(t-1)})$ for each $i \in \overline{S^{(t-1)}}$, which is an *old* marginal gain, i.e., $\rho_i = f_i(S^{(u_i)})$ for some $u_i \leq t - 1$. In each

[1] Strictly speaking, this is the so-called $L$-ensemble DPP, but we here call it a DPP for simplicity.

**Algorithm 1** FASTGREEDY [13] for cardinality-constrained DPP MAP inference

1: $V \leftarrow O,\ d_i^{(0)} \leftarrow \sqrt{L_{i,i}}\ (\forall i \in [n]),\ S^{(0)} \leftarrow \emptyset$
2: **for** $t = 1$ to $k$ **do**
3: Take $j_t \in \arg\max_{i \in \overline{S^{(t-1)}}} d_i^{(t-1)}$ ▷ Terminate if $d_{j_t}^{(t-1)} \leq 1$ (i.e., $f_{j_t}(S^{(t-1)}) \leq 0$)
4: $S^{(t)} \leftarrow S^{(t-1)} \cup \{j_t\}$ ▷ $S^{(t)} = \{j_1, \dots, j_t\}$
5: **for** $i$ in $\overline{S^{(t)}}$ **do** ▷ Skip Lines 5–7 (updates for the next step) if $t = k$
6: $V_{i,j_t} \leftarrow (L_{i,j_t} - \langle V[i, S^{(t-1)}], V[j_t, S^{(t-1)}] \rangle) / d_{j_t}^{(t-1)}$ ▷ $V[i, \emptyset] = \mathbf{0}$ for any $i \in [n]$
7: $d_i^{(t)} \leftarrow \sqrt{\left(d_i^{(t-1)}\right)^2 - {V_{i,j_t}}^2}$
8: **return** $S^{(k)}$ ▷ Return $S^{(t-1)}$ if terminates with $t < k$

iteration, LAZYGREEDY picks $i \in \overline{S^{(t-1)}}$ with the largest $\rho_i$ value as the most promising element. It then updates the $\rho_i$ value to the latest marginal gain $f_i\left(S^{(t-1)}\right)$. If $\rho_i$ is still the largest among $\rho_{i'}$ for all $i' \in \overline{S^{(t-1)}}$, the diminishing returns property guarantees $i \in \arg\max\{ f_{i'}(S^{(t-1)}) \mid i' \in \overline{S^{(t-1)}} \}$, and hence it adds $j_t = i$ to $S^{(t-1)}$. LAZYGREEDY thus constructs a greedy solution, deferring updates of upper bounds of unpromising elements. If the upper bounds are managed by a priority queue, every single iteration computes the marginal gain only once and makes $\mathrm{O}(\log n)$ comparisons (note that "iteration" is distinguished from "step" of GREEDY). With this contrivance, LAZYGREEDY runs much faster than GREEDY in practice, even though it does not improve the worst-case complexity.

### 2.2 Fast greedy algorithm for DPP MAP inference

We turn to DPP MAP inference with a kernel matrix $L = B^\top B$ and $B \in \mathbb{R}^{d \times n}$. If we apply GREEDY to $f(S) = \log\det L[S]$, it takes $\mathrm{O}(k^{\omega+1}n)$ time since computing an $f$ value takes $\mathrm{O}(k^\omega)$ time; in addition, computing $L$ at first takes $\mathrm{O}(\min\{n^{\omega-1}d, n^2 d^{\omega-2}\})$ time. FASTGREEDY [13] provides an $\mathrm{O}(knd)$-time implementation of GREEDY for cardinality-constrained DPP MAP inference.

Algorithm 1 describes the procedure of FASTGREEDY, which is based on the Cholesky factorization of $L$ with maximum pivoting. The Cholesky decomposition produces a matrix $V \in \mathbb{R}^{n \times n}$ such that $L = VV^\top$ and $PV$ is lower triangular for some permutation matrix $P$. We call $V$ a *Cholesky factor* of $L$. In each $t$th step, Lines 5–7 calculate the $t$th column of $PV$ via backward substitution (see Fig. 1a). The following Proposition 2.1 implies an important fact that once $V[i, S^{(t)}]$ is filled, we can obtain $f_i(S^{(t)})$ from $d_i^{(t)}$ computed in Line 7, which is equal to the diagonal $V_{i,i}$ of the current Cholesky factor. Although the proposition is already proved in [13], we present a proof sketch since it would be helpful to understand the subsequent discussion (see [13] for the complete proof).

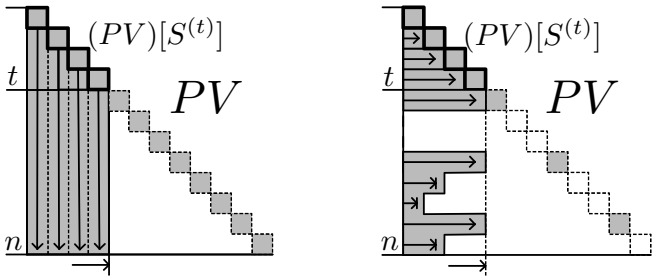
**Proposition 2.1** ([13]). *For $t = 0, 1, \dots, k-1$, it holds that $f_i\left(S^{(t)}\right) = 2\log d_i^{(t)}$ for every $i \in \overline{S^{(t)}}$.*

*Proof sketch.* The proof is by induction on $t$. If $t = 0$, it holds that $f_i\left(S^{(t)}\right) = f(\{i\}) - f(\emptyset) = \log L_{i,i} = 2\log d_i^{(0)}$ for $i \in [n]$. Given a Cholesky factor $V[S^{(t)}]$ of $L[S^{(t)}]$, for $i \in \overline{S^{(t)}}$, we have

$$L[S^{(t)} \cup \{i\}] = \begin{bmatrix} L[S^{(t)}] & L[S^{(t)}, i] \\ L[i, S^{(i)}] & L_{i,i} \end{bmatrix} = \begin{bmatrix} V[S^{(t)}] & \mathbf{0} \\ V[i, S^{(t)}] & d_i^{(t)} \end{bmatrix} \begin{bmatrix} V[S^{(t)}]^\top & V[i, S^{(t)}]^\top \\ \mathbf{0}^\top & d_i^{(t)} \end{bmatrix}.$$

Hence, we have $\log\det L[S^{(t)} \cup \{i\}] = \log\left(d_i^{(t)} \det V[S^{(t)}]\right)^2 = 2\log d_i^{(t)} + \log\det L[S^{(t)}]$, which implies $\log\det L[S^{(t)} \cup \{i\}] - \log\det L[S^{(t)}] = 2\log d_i^{(t)}$. Thus, the statement holds. □

Therefore, by iteratively adding $j_t \in \arg\max_{i \in \overline{S^{(t-1)}}} d_i^{(t-1)}$ to $S^{(t-1)}$ as in Algorithm 1, we can obtain a greedy solution. Since $L_{i,j_t} = \langle \boldsymbol{\phi}_i, \boldsymbol{\phi}_{j_t} \rangle$ is computed in $\mathrm{O}(d)$ time and $d \geq k$, Line 6 takes $\mathrm{O}(d)$ time. This is repeated $\mathrm{O}(kn)$ times, hence the total time complexity of $\mathrm{O}(knd)$.




(a) FastGreedy (b) LazyFastGreedy

Figure 1: Images of Cholesky factors computed by FastGreedy and LazyFastGreedy. Off-diagonals are shaded if they are already computed, and arrows represent the direction in which the computation of off-diagonals proceeds. Diagonals are shaded if they give the latest marginal gains, each of which becomes available once $|S^{(t)}|$ off-diagonals in the same row are computed (shaded). Diagonals with bold lines correspond to elements that are already selected.

## 3 Lazy and fast algorithms for cardinality-constrained DPP MAP inference

### 3.1 Lazy and fast greedy algorithm

We combine LazyGreedy and FastGreedy to obtain even faster LazyFastGreedy.

As described above, LazyGreedy is designed for submodular function maximization under the oracle model, where a marginal gain is computed for each $i \in \overline{S^{(t-1)}}$. Meanwhile, the core idea of FastGreedy is to obtain marginal gains of all $i \in \overline{S^{(t-1)}}$ efficiently by computing a new column of a Cholesky factor, as in Fig. 1a. To combine these seemingly incompatible methods, we need to take a closer look at how FastGreedy updates the entries in $V$. The following observation is obvious but elucidates the essential row-wise independence of the updates of Cholesky-factor entries.

**Observation 3.1.** *In each $t$th step in Algorithm 1, for each $i \in \overline{S^{(t)}}$, $V_{i,j_t}$ in Line 6 is computed from $d_{j_t}^{(t-1)}$, $L_{i,j_t}$, and $V[\{i, j_t\}, S^{(t-1)}]$. Thus, conditioned on $S^{(t)}$, the $i$th row of $V[\overline{S^{(t)}}, S^{(t)}]$ can be updated independently for each $i \in \overline{S^{(t)}}$ in a sense that computing $V_{i,j_t}$ only requires $V_{i,j_{t'}}$ for $t' < t$ and $V[j_t, S^{(t-1)}]$, which is included in the already fixed Cholesky factor $V[S^{(t)}]$.*

That is, in Fig. 1b, once $(PV)[S^{(t)}]$ is fixed, we can update the $i$th row independently for each $i \in \overline{S^{(t)}}$. In the words of submodular function maximization, we can compute $f_i\big(S^{(t)}\big)$ if $f_{j_{t'}}\big(S^{(t'-1)}\big)$ and $f_i\big(S^{(t'-1)}\big)$ for $t' \leq t$ are available. This enables us to apply the idea of lazy updates to FastGreedy.

Algorithm 2 describes our LazyFastGreedy, and Fig. 1b illustrates how entries in $V$ are updated. In each iteration, it picks the most promising element $i$ in Line 3, as with LazyGreedy. Then, it calls UpdateRow to compute the entries of $V[i, S]$ via backward substitution. Once $V[i, S]$ is filled, the resulting $d_i$ computed in Line 12 satisfies $2 \log d_i = f_i(S)$ by Proposition 2.1, thus obtaining the latest marginal gain of $i$. Lines 5–7 check whether the latest $f_i(S)$ is still the largest among the (old) marginal gains of $i' \in \overline{S}$; if so, $j_{|S|+1} = i$ is added to $S$. Note that the deferred updates of $d_{i'}$ ($i' \in \overline{S} \setminus \{i\}$) do not matter when deciding whether to add $i$ to $S$ due to the submodularity. Here, $\boldsymbol{d} = (d_1, \ldots, d_n)$ plays the role of upper bounds $(\rho_1, \ldots, \rho_n)$ of LazyGreedy and is maintained by a priority queue. LazyFastGreedy thus finds a greedy solution by exactly mimicking the behavior of LazyGreedy while computing marginal gains efficiently as with FastGreedy.

The vector $\boldsymbol{u} = (u_1, \ldots, u_n)$ keeps track of when the upper bounds are last updated; specifically, each $u_i \in \{0, 1, \ldots, k-1\}$ indicates that the upper bound $d_i$ is last updated with respect to $\{j_1, \ldots, j_{u_i}\}$, i.e., $2 \log d_i = f_i(\{j_1, \ldots, j_{u_i}\})$. Since UpdateRow starts to fill $V[i, S]$ from the $(u_i + 1)$st entry, no off-diagonals $V_{i,j_t}$ are computed more than once in Line 11. Thus, the complexity of Algorithm 2 depends on the number of computed off-diagonals. We denote it by $u = \sum_{i \in [n]} u_i$ with $u_i$ values at the end of Algorithm 2. The $u$ value changes in the range of $[k(k-1)/2, (k-1)(n-k/2)]$ depending on how well the lazy update works and affects the overall time complexity as follows.

**Algorithm 2** LAZYFASTGREEDY for cardinality-constrained DPP MAP inference

1: $V \leftarrow O, \boldsymbol{d} \leftarrow \left(\sqrt{L_{i,i}}\right)_{i\in[n]}, \boldsymbol{u} \leftarrow \mathbf{0}, S \leftarrow \emptyset$ ▷ $\boldsymbol{d}$ is maintained by a priority queue
2: **while** $|S| < k$ **do**
3: Take $i \in \arg\max_{i'\in\overline{S}} d_{i'}$ ▷ Terminate if $d_i \leq 1$ (i.e., $f_i(S) \leq 0$)
4: UPDATEROW$(V, \boldsymbol{d}, \boldsymbol{u}; i, S, L)$ ▷ Nothing is done if $|S| = 0$
5: **if** $d_i \geq \max_{i'\in\overline{S}} d_{i'}$ **then** ▷ Otherwise insert $d_i$ into the priority queue
6: $j_{|S|+1} \leftarrow i$
7: $S \leftarrow S \cup \{j_{|S|+1}\}$
8: **return** $S$

9: **function** UPDATEROW$(V, \boldsymbol{d}, \boldsymbol{u}; i, S, L)$ ▷ $S = \left\{j_1, j_2, \ldots, j_{|S|}\right\}$
10: **for** $t = u_i + 1, u_i + 2, \ldots, |S|$ **do**
11: $V_{i,j_t} \leftarrow (L_{i,j_t} - \langle V[i, S^{(t-1)}], V[j_t, S^{(t-1)}]\rangle)/d_{j_t}$ ▷ $S^{(t-1)} = \{j_1, \ldots, j_{t-1}\}$
12: $d_i \leftarrow \sqrt{{d_i}^2 - {V_{i,j_t}}^2}$
13: $u_i \leftarrow |S|$ ▷ This line is not needed in Algorithm 3

**Theorem 3.2.** *Algorithm 2 returns a greedy solution in* $\mathrm{O}(nd + u(d + \log n))$ *time. If the lazy update works best and worst, it runs in* $\mathrm{O}((n + k^2)d)$ *and* $\mathrm{O}(kn(d + \log n))$ *time, respectively.*

*Proof.* Algorithm 2 returns a greedy solution as explained above. We below discuss the running time.

At the beginning, we need $\mathrm{O}(nd)$ time to compute $L_{i,i} = \langle \boldsymbol{\phi}_i, \boldsymbol{\phi}_i \rangle$ for $i = 1, \ldots, n$. In UPDATEROW, an access to $L_{i,j_t}$ in Line 11 takes $\mathrm{O}(d)$ time, and the inner product takes $\mathrm{O}(k)$ ($\lesssim \mathrm{O}(d)$) time. This computation is done $u$ times, and thus the total computation time caused by UPDATEROW is $\mathrm{O}(ud)$. In Line 5, we need $\mathrm{O}(\log n)$ time to update the priority queue if $d_i < \max_{i'\in\overline{S}} d_{i'}$, which can hold only when at least one off-diagonal is computed in UPDATEROW. Therefore, Line 5 takes $\mathrm{O}(u \log n)$ time in total. Thus, the overall time complexity is $\mathrm{O}(nd + u(d + \log n))$.

Let $S$ be the output of Algorithm 2 and $P$ a permutation matrix such that $(PV)[S]$ is lower triangular. In the best case, UPDATEROW is called up to $k$ times and $u = k(k-1)/2$ off-diagonals of $(PV)[S]$ are computed. Moreover, updates of the priority queue are done only up to $k$ times, taking $\mathrm{O}(k \log n)$ ($\lesssim \mathrm{O}(nd)$) time in total. Thus, it runs in $\mathrm{O}((n+k^2)d)$ time. In the worst case, Algorithm 2 calculates the off-diagonals of $(PV)[S]$ and all the entries of $V[\overline{S}, S \setminus \{j_k\}]$; the total number of those entries is $u = k(k-1)/2 + (k-1)(n-k) = (k-1)(n-k/2)$. Hence, it takes $\mathrm{O}(kn(d + \log n))$ time. □

The best-case time complexity is better than $\mathrm{O}(knd)$ of FASTGREEDY if $k = \mathrm{o}(n)$. Even in the worse case, it is as fast as FASTGREEDY if $d = \Omega(\log n)$. Note that both $k = \mathrm{o}(n)$ and $d = \Omega(\log n)$ are true in most practical situations. Experiments in Section 5 demonstrate that LAZYFASTGREEDY can run much faster than FASTGREEDY in practice.

### 3.2 Extension to random, stochastic, and interlace greedy algorithms

The core idea of LAZYFASTGREEDY can be used for speeding up other greedy-type algorithms: RANDOMGREEDY [10], STOCHASTICGREEDY [38, 43], and INTERLACEGREEDY [28], which enjoy $1/\mathrm{e}$-, $1/4$-, and $1/4$-approximation guarantees, respectively, for non-monotone submodular function maximization with a cardinality constraint. Note that the guarantees for the non-monotone case are essential in DPP MAP inference since the log-determinant function is non-monotone in general. Due to the space limitation, we present the details of those extensions in Appendix A.

## 4 Fast double greedy algorithm for unconstrained DPP MAP inference

This section discusses unconstrained DPP MAP inference with a kernel matrix $L = B^\top B$, where $B \in \mathbb{R}^{d\times n}$. In this setting, if $f(S) = \log\det L[S]$ is monotone, $S = [n]$ is a trivial optimal solution. Thus, we suppose $f$ to be non-monotone. We also assume $L$ to be positive definite since the algorithm of [11] discussed below requires $f(S) > -\infty$ for any $S \subseteq [n]$. Note that this implies $d \geq n$.

**Algorithm 3** FASTDOUBLEGREEDY for unconstrained DPP MAP inference

1: Compute $L = B^\top B$ and $L^{-1}$
2: $V \leftarrow O, W \leftarrow O, \boldsymbol{d} \leftarrow \left(\sqrt{L_{i,i}}\right)_{i\in[n]}, \boldsymbol{e} \leftarrow \left(\sqrt{(L^{-1})_{i,i}}\right)_{i\in[n]}, S \leftarrow \emptyset$
3: **for** $i = 1$ to $n$ **do**
4: UPDATEROW$(V, \boldsymbol{d}, \mathbf{0}; i, S, L)$ ▷ $S$ is sorted in order of $1, 2, \ldots, n$
5: UPDATEROW$(W, \boldsymbol{e}, \mathbf{0}; i, [i] \setminus S, L^{-1})$ ▷ $[i] \setminus S$ is sorted in order of $1, 2, \ldots, n$
6: $a_i \leftarrow \max\{2 \log d_i, 0\}, b_i \leftarrow \max\{2 \log e_i, 0\}$
7: $S \leftarrow S \cup \{i\}$ w.p. $a_i/(a_i + b_i)$ ▷ Implicity update $T = \overline{[n] \setminus S}$ w.p. $b_i/(a_i + b_i)$
8: **return** $S$

A famous algorithm for unconstrained submodular function maximization is DOUBLEGREEDY [11], a randomized $1/2$-approximation algorithm. Although it calls an evaluation oracle only $\mathrm{O}(n)$ times, its naive implementation is too costly for large DPP MAP inference instances since computing the log-determinant function value takes $\mathrm{O}(n^\omega)$ time, which will lead to the total time of $\mathrm{O}(n^{\omega-1}d + n^{\omega+1})$. We below extend the idea of FASTGREEDY [13] to DOUBLEGREEDY and obtain its $\mathrm{O}(n^{\omega-1}d + n^3)$-time implementation for unconstrained DPP MAP inference.

DOUBLEGREEDY maintains two subsets $S$ and $T$, which are initially set to $S = \emptyset$ and $T = [n]$. For $i = 1, \ldots, n$, it computes $a_i = \max\{f_i(S), 0\}$ and $b_i = \max\{-f_i(T \setminus \{i\}), 0\}$, and then either adds $i$ to $S$ with probability $a_i/(a_i + b_i)$ or removes $i$ from $T$ with probability $b_i/(a_i + b_i)$.[2] Note that $T = [n] \setminus ([i] \setminus S) = \overline{[i] \setminus S}$ always holds. Finally, it returns $S$ (or equivalently $T = \overline{[n] \setminus S} = S$).

As for the growing subset $S$, we can efficiently compute marginal gains $f_i(S)$ by incrementally updating a Cholesky factor, as with FASTGREEDY. When it comes to the shrinking subset $T$, however, we cannot directly use the efficient incremental update for computing $-f_i(T \setminus \{i\})$. If we naively compute it in each step, it takes $\mathrm{O}(n^\omega)$ time, resulting in the same time complexity as the naive implementation. Our key idea for overcoming this difficulty is to use the following Jacobi's complementary minor formula (see, e.g., [9]).

**Proposition 4.1.** *Let $L \in \mathbb{R}^{n\times n}$ be a non-singular matrix and $I, J \subseteq [n]$ be subsets with $|I| = |J|$. Then, it holds that* $\det L[I, J] = (-1)^{\sum_{i\in I} i + \sum_{j\in J} j} \det L \det L^{-1}[\overline{I}, \overline{J}]$.

This formula provides a lemma that enables us to compute $-f_i(T \setminus \{i\})$ via incremental updates.

**Lemma 4.2.** *Let $L \in \mathbb{R}^{n\times n}$ be positive definite. For any $S \subseteq [n]$, define $f(S) = \log\det L[S]$ and $g(S) = \log\det L^{-1}[S]$. Then, $g(S \cup \{i\}) - g(S) = f(\overline{S} \setminus \{i\}) - f(\overline{S})$ holds for any $S \subseteq [n]$ and $i \in [n]$.*

*Proof.* By using Proposition 4.1, we can prove the claim as follows:

$$\begin{aligned} f(\overline{S} \setminus \{i\}) - f(\overline{S}) &= \log\det L[\overline{S} \setminus \{i\}] - \log\det L[\overline{S}] \\ &= \log\det L \det L^{-1}[[n] \setminus (\overline{S} \setminus \{i\})] - \log\det L \det L^{-1}[[n] \setminus \overline{S}] \\ &= \log\det L^{-1}[S \cup \{i\}] - \log\det L^{-1}[S] \\ &= g(S \cup \{i\}) - g(S). \end{aligned}$$

□

Lemma 4.2 implies $g(([i] \setminus S) \cup \{i\}) - g([i] \setminus S) = f(\overline{([i] \setminus S)} \setminus \{i\}) - f(\overline{[i] \setminus S}) = -f_i(T)$. Therefore, by computing $L = B^\top B$ and $L^{-1}$ in $\mathrm{O}(n^{\omega-1}d)$ ($\gtrsim \mathrm{O}(n^\omega)$) time in advance, we can compute $-f_i(T \setminus \{i\})$ in each step by incrementally updating a Cholesky factor of size $|[i] \setminus S|$.

Algorithm 3 presents our FASTDOUBLEGREEDY based on this idea. In each $i$th step, the first and second calls to UPDATEROW, defined in Algorithm 2, fill $V[i, S]$ and $W[i, [i] \setminus S]$, respectively. Hence, Proposition 2.1 and Lemma 4.2 imply $2 \log d_i = f_i(S)$ and $2 \log e_i = g_i([i] \setminus S) = -f_i(T)$. Thus, Algorithm 3 exactly mimics the behavior of DOUBLEGREEDY while efficiently computing marginal gains via incremental updates of Cholesky factors. In UPDATEROW defined in Algorithm 2, since $L$ and $L^{-1}$ are already computed, each $V_{i,j_t}$ is calculated in $\mathrm{O}(n)$ time; therefore, a single call to UPDATEROW takes $\mathrm{O}(n^2)$ time. Since UPDATEROW is called $2n$ times, Lines 2–7 construct a solution in $\mathrm{O}(n^3)$ time. In total, Algorithm 3 runs in $\mathrm{O}(n^{\omega-1}d + n^3)$ time.

[2] The algorithm thus sequentially examines all elements, and hence there is no room for the lazy update.

## 5 Experiments

We evaluate the effectiveness of our acceleration techniques on synthetic and real-world datasets. Section 5.1 examines speed-ups of GREEDY for cardinality-constrained DPP MAP inference, and Section 5.2 focuses on DOUBLEGREEDY for the unconstrained setting. We present experimental results on RANDOMGREEDY, STOCHASTICGREEDY, and INTERLACEGREEDY in Appendix A.4.

The algorithms are implemented in C++ with library Eigen 3.4.0 for matrix computations. Experiments are conducted using a compiler GCC 10.2.0 on a computer with 3.8 GHz Intel Xeon Gold CPU and 800 GB RAM.

We use synthetic and two real-world datasets, Netflix Prize [5] and MovieLens [25]. Each dataset provides a matrix $B \in \mathbb{R}^{d \times n}$ consisting of column vectors $\boldsymbol{\phi}_1, \boldsymbol{\phi}_2, \dots, \boldsymbol{\phi}_n \in \mathbb{R}^d$ of $n$ items, which defines an $n \times n$ kernel matrix $L = B^\top B$. Below we explain the item vectors of each dataset.

**Synthetic datasets.** We use the setting of [21]. Each entry of $\boldsymbol{\phi}_i \in \mathbb{R}^d$ is independently drawn from the standard normal distribution, $\phi_{ij} \sim \mathcal{N}(0, 1)$. As a result, the kernel matrix $L$ conforms to a Wishart distribution with $n$ degrees of freedom and an identity covariance matrix, i.e., $L \sim \mathcal{W}(I, n, n)$. We consider various $n$ values in the experiments below, and we always set the vector length $d$ to $n$.

**Real-world datasets.** Both Netflix Prize and MovieLens datasets contain users' ratings of movies from one to five, where we regard a movie as an item. Following [13], we binarize the ratings based on whether it is greater than or equal to four. After that, we eliminate movies that result in all-zero vectors and users who result in all-zero ratings since those are redundant. Consequently, the Netflix Prize dataset has $n = 17770$ movies and $d = 478615$ users with 56919190 ratings; the MovieLens dataset has $n = 40858$ movies and $d = 162342$ users with 12452811 ratings.

**$B$- and $L$-input settings.** It is important to care about whether $L = B^\top B$ is computed in advance or not. In practice, a matrix $B \in \mathbb{R}^{d \times n}$ of item vectors is often given. Then, computing $L = B^\top B$ in advance takes $\mathrm{O}(\min\{n^{\omega-1}d, n^2 d^{\omega-2}\})$ time, which we should avoid when $n$ is large since the running time of LAZYFASTGREEDY (and FASTGREEDY) increases only linearly in $n$. On the other hand, we are sometimes given a pre-computed kernel matrix $L$, and we can access $L_{i,j}$ in $\mathrm{O}(1)$ time.

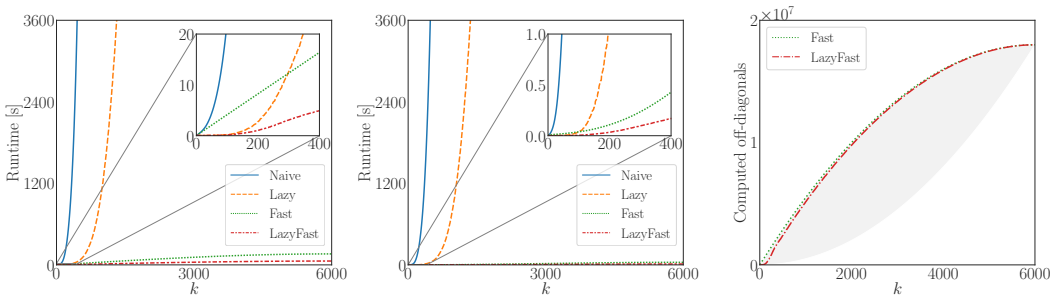
We below consider both settings, called $B$- and $L$-input settings, respectively. In the $L$-input setting, we exclude the time to compute $L = B^\top B$ from consideration. Under this condition, FASTGREEDY takes $\mathrm{O}(k^2 n)$ time and LAZYFASTGREEDY does $\mathrm{O}(n + u(k + \log n))$, where the first $\mathrm{O}(n)$ term is for constructing a priority queue. By similar reasoning to that in the proof of Theorem 3.2, it runs in $\mathrm{O}(n + k^3 + k \log n)$ and $\mathrm{O}(kn(k + \log n))$ time if the lazy update works best and worst, respectively.

### 5.1 Greedy algorithm for cardinality-constrained DPP MAP inference

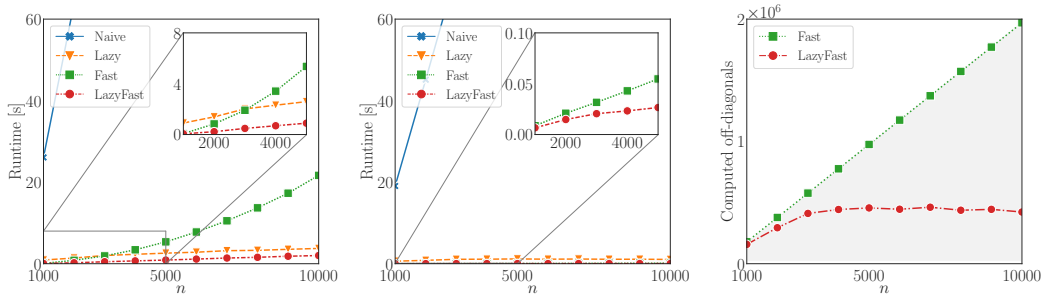
We compare the running time of GREEDY, LAZYGREEDY, FASTGREEDY, and LAZYFASTGREEDY, which we here call Naive, Lazy, Fast, and LazyFast, respectively, for short. As regards synthetic datasets, we consider two settings that fix either $n$ or $k$: (i) $n = 6000$ and $k = 1, 2, \dots, n$, and (ii) $k = 200$ and $n = 1000, 2000, \dots, 10000$. We set the timeout periods of (i) and (ii) to 3600 and 60 seconds, respectively. Regarding real-world datasets, $n$ is fixed as explained above and we increase $k = 1, 2, \dots, n$. We set the timeout period to 3600 seconds. With the Netflix (MovieLens) dataset, the objective value peaked with $k = 17762$ ($k = 18763$), and thus we stopped increasing $k$ after that.

Figures 2 and 3 present the results on synthetic and real-world datasets, respectively. The curves in the runtime figures represent that faster algorithms can return greedy solutions to instances with larger $k$ or $n$ values within the timeout periods. The rightmost figures present the number of off-diagonals of Cholesky factors computed by Fast and LazyFast. As explained in Section 3.1, while Fast always computes all the off-diagonals of $V[[n], S]$, LazyFast does not due to the lazy update.

LazyFast was the fastest in all the settings. In particular, in the synthetic (ii) and real-world settings, LazyFast computed fewer off-diagonals than Fast, thus running the fastest. In the synthetic setting (i), although LazyFast computed almost all off-diagonals in $V[[n], S]$, it was still faster than Fast. For example, for the $L$-input setting with $n = k = 6000$, Fast and LazyFast took 37.4 and 13.7 seconds, respectively. This unexpected speed-up is caused by the cache efficiency of LazyFast. Specifically, every call to UPDATEROW computes off-diagonals from $V_{i,j_{u_i+1}}$ to $V_{i,j_{|S|}}$ by accessing entries only in




(a) $n = 6000$, $B$-input, Runtime (b) $n = 6000$, $L$-input, Runtime (c) $n = 6000$, Off-diagonals




(d) $k = 200$, $B$-input, Runtime (e) $k = 200$, $L$-input, Runtime (f) $k = 200$, Off-diagonals

Figure 2: Results on synthetic datasets. In the four runtime figures, enlarged views of lower left parts are shown for visibility. In Figs. 2c and 2f, the gray band indicates the range of the possible number of computed off-diagonals: $[k(k-1)/2, (k-1)(n-k/2)]$.

$V[\{i, j_{u_i+1}, \ldots, j_{|S|}\}, S^{(|S|-1)}]$. This process virtually creates blocks in a Cholesky factor, enabling the cache-efficient computation of off-diagonals via blocking (see, e.g., [26, Section 2.3]). In fact, the cache miss rates of Fast and LazyFast for the above example were 71.3% and 3.3%, respectively.

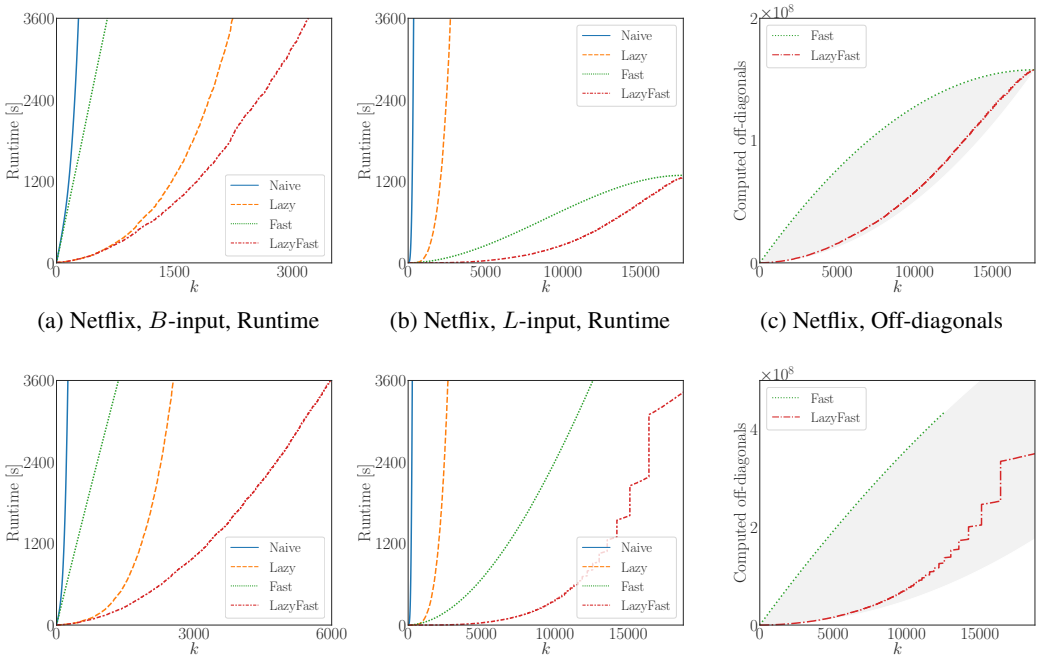
Another interesting observation in the synthetic (ii) and real-world settings is that while Fast was often faster than Lazy in the $L$-input setting, the opposite occurred in the $B$-input setting. This is because computing an off-diagonal in the $B$-input setting is costly relative to the $L$-input setting. As a result, avoiding the redundant computation of off-diagonals by the lazy update tends to be more effective than computing marginal gains efficiently via the Cholesky factorization.

### 5.2 Double greedy algorithm for unconstrained DPP MAP inference

We compare naive DOUBLEGREEDY and our FASTDOUBLEGREEDY by applying them to unconstrained DPP MAP inference on three datasets: synthetic ($n = 6000$), Netflix Prize, and MovieLens. Since kernel matrices $L$ in the real-world datasets are singular, we use kernel matrices computed as $L = 0.9B^\top B + 0.1I$ in this section, ensuring that the resulting matrices $L$ are positive definite. In this section, we set the timeout period to one day (86400 seconds).

Table 1 presents the results. Note that both algorithms require $L$ to be computed in advance. Therefore, we measured the time of computing $B^\top B$ (Product) separately from the time of constructing solutions (Greedy). Our FASTDOUBLEGREEDY additionally requires $L^{-1}$ to be computed in advance for accelerating the solution construction; therefore, we also measured the time of computing $L^{-1}$ (Inverse) separately. As in Table 1, naive DOUBLEGREEDY took so long for solution construction that it failed to return solutions to real-world instances in one day. By contrast, our FASTDOUBLEGREEDY constructed solutions far faster and succeeded in returning solutions to all the instances.

Also, the computation of $B^\top B$ (Product) took a considerably long time relative to the running time of LAZYFASTGREEDY in the previous section. Therefore, as mentioned above, when a matrix $B$ of item vectors is given and our goal is to obtain a greedy solution for small $k = o(n)$, we should avoid computing the kernel matrix $L = B^\top B$ in advance.




(a) Netflix, $B$-input, Runtime (b) Netflix, $L$-input, Runtime (c) Netflix, Off-diagonals

(d) MovieLens, $B$-input, Runtime (e) MovieLens, $L$-input, Runtime (f) MovieLens, Off-diagonals

Figure 3: Results on real-world datasets. In Figs. 3c and 3f, the gray band indicates the range of the possible number of computed off-diagonals: $[k(k-1)/2, (k-1)(n-k/2)]$.

Table 1: Running time [s] of DOUBLEGREEDY and FASTDOUBLEGREEDY

| Dataset | DOUBLEGREEDY | | | FASTDOUBLEGREEDY | | | |
|---|---|---|---|---|---|---|---|
| | Product | Greedy | **Total** | Product | Inverse | Greedy | **Total** |
| Synthetic | | | | | | | |
| $n = 2000$ | 1.1 | 258.6 | 259.7 | 1.1 | 1.8 | 0.7 | 3.6 |
| $n = 4000$ | 8.7 | 5422.4 | 5431.1 | 8.7 | 16.9 | 10.0 | 35.6 |
| $n = 6000$ | 30.0 | 36233.1 | 36263.1 | 30.0 | 59.8 | 34.6 | 124.4 |
| $n = 8000$ | 70.2 | $> 86400.0$ | — | 70.2 | 151.2 | 103.1 | 324.5 |
| $n = 10000$ | 137.9 | $> 86400.0$ | — | 137.9 | 294.2 | 182.6 | 614.7 |
| Netflix Prize | 20561.6 | $> 86400.0$ | — | 20561.6 | 1706.7 | 916.2 | 23184.5 |
| MovieLens | 37829.5 | $> 86400.0$ | — | 37829.5 | 21999.2 | 6337.3 | 66166.0 |


## Acknowledgements

The authors thank anonymous reviewers for their valuable comments. This work was supported by JST ERATO Grant Number JPMJER1903 and JSPS KAKENHI Grant Number JP22K17853.


## References


[1] N. Anari and M. Dereziński. Isotropy and log-concave polynomials: Accelerated sampling and high-precision counting of matroid bases. In *Proceedings of the 2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS 2020)*, pages 1331–1344. IEEE, 2020.

[2] N. Anari, S. Oveis Gharan, and A. Rezaei. Monte Carlo Markov chain algorithms for sampling strongly Rayleigh distributions and determinantal point processes. In *Proceedings of the 29th Annual Conference on Learning Theory (COLT 2016)*, volume 49, pages 103–115. PMLR, 2016.

[3] E. Balkanski, A. Breuer, and Y. Singer. Non-monotone submodular maximization in exponentially fewer iterations. In *Advances in Neural Information Processing Systems (NeurIPS 2018)*, volume 31. Curran Associates, Inc., 2018.

[4] E. Balkanski, A. Rubinstein, and Y. Singer. An exponential speedup in parallel running time for submodular maximization without loss in approximation. In *Proceedings of the 2019 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2019)*, pages 283–302. SIAM, 2019.

[5] J. Bennett and S. Lanning. The Netflix prize. In *Proceedings of KDD Cup and Workshop*, 2007. URL https://www.kaggle.com/datasets/netflix-inc/netflix-prize-data.

[6] A. Bhaskara, A. Karbasi, S. Lattanzi, and M. Zadimoghaddam. Online MAP inference of determinantal point processes. In *Advances in Neural Information Processing Systems (NeurIPS 2020)*, volume 33, pages 3419–3429. Curran Associates, Inc., 2020.

[7] D. Bini and V. Y. Pan. *Polynomial and Matrix Computations*, volume 1. Birkhäuser Boston, 1994.

[8] A. Breuer, E. Balkanski, and Y. Singer. The FAST algorithm for submodular maximization. In *Proceedings of the 37th International Conference on Machine Learning (ICML 2020)*, volume 119, pages 1134–1143. PMLR, 2020.

[9] R. A. Brualdi and H. Schneider. Determinantal identities: Gauss, Schur, Cauchy, Sylvester, Kronecker, Jacobi, Binet, Laplace, Muir, and Cayley. *Linear Algebra and Its Applications*, 52-53:769–791, 1983.

[10] N. Buchbinder, M. Feldman, J. S. Naor, and R. Schwartz. Submodular maximization with cardinality constraints. In *Proceedings of the 2014 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2014)*, pages 1433–1452. SIAM, 2014.

[11] N. Buchbinder, M. Feldman, J. S. Naor, and R. Schwartz. A tight linear time $(1/2)$-approximation for unconstrained submodular maximization. *SIAM Journal on Computing*, 44(5):1384–1402, 2015.

[12] D. Calandriello, M. Derezinski, and M. Valko. Sampling from a $k$-DPP without looking at all items. In *Advances in Neural Information Processing Systems (NeurIPS 2020)*, volume 33, pages 6889–6899. Curran Associates, Inc., 2020.

[13] L. Chen, G. Zhang, and E. Zhou. Fast greedy MAP inference for determinantal point process to improve recommendation diversity. In *Advances in Neural Information Processing Systems (NeurIPS 2018)*, volume 31. Curran Associates, Inc., 2018.

[14] Y. Chen, T. Dey, and A. Kuhnle. Best of both worlds: Practical and theoretically optimal submodular maximization in parallel. In *Advances in Neural Information Processing Systems (NeurIPS 2021)*, volume 34, pages 25528–25539. Curran Associates, Inc., 2021.

[15] A. Çivril and M. Magdon-Ismail. On selecting a maximum volume sub-matrix of a matrix and related problems. *Theoretical Computer Science*, 410(47):4801–4811, 2009.

[16] M. Derezinski, D. Calandriello, and M. Valko. Exact sampling of determinantal point processes with sublinear time preprocessing. In *Advances in Neural Information Processing Systems (NeurIPS 2019)*, volume 32. Curran Associates, Inc., 2019.

[17] A. Ene and H. Nguyen. Parallel algorithm for non-monotone DR-submodular maximization. In *Proceedings of the 37th International Conference on Machine Learning (ICML 2020)*, volume 119, pages 2902–2911. PMLR, 2020.

[18] M. Fahrbach, V. Mirrokni, and M. Zadimoghaddam. Non-monotone submodular maximization with nearly optimal adaptivity and query complexity. In *Proceedings of the 36th International Conference on Machine Learning (ICML 2019)*, volume 97, pages 1833–1842. PMLR, 2019.

[19] K. Fan. An inequality for subadditive functions on a distributive lattice, with application to determinantal inequalities. *Linear Algebra and Its Applications*, 1(1):33–38, 1968.

[20] M. Gartrell, I. Han, E. Dohmatob, J. Gillenwater, and V.-E. Brunel. Scalable learning and MAP inference for nonsymmetric determinantal point processes. In *Proceedings of the 9th International Conference on Learning Representations (ICLR 2021)*, 2021.

[21] J. Gillenwater, A. Kulesza, and B. Taskar. Near-optimal MAP inference for determinantal point processes. In *Advances in Neural Information Processing Systems (NeurIPS 2012)*, volume 25. Curran Associates, Inc., 2012.

[22] J. Gillenwater, A. Kulesza, Z. Mariet, and S. Vassilvtiskii. A tree-based method for fast repeated sampling of determinantal point processes. In *Proceedings of the 36th International Conference on Machine Learning (ICML 2019)*, volume 97, pages 2260–2268. PMLR, 2019.

[23] I. Han and J. Gillenwater. MAP inference for customized determinantal point processes via maximum inner product search. In *Proceedings of the 23rd International Conference on Artificial Intelligence and Statistics (AISTATS 2020)*, volume 108, pages 2797–2807. PMLR, 2020.

[24] I. Han, P. Kambadur, K. Park, and J. Shin. Faster greedy MAP inference for determinantal point processes. In *Proceedings of the 34th International Conference on Machine Learning (ICML 2017)*, volume 70, pages 1384–1393. PMLR, 2017.

[25] F. M. Harper and J. A. Konstan. The MovieLens datasets: History and context. *ACM Transactions on Interactive Intelligent Systems*, 5(4):1–19, 2015. URL https://grouplens.org/datasets/movielens/25m/.

[26] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier, 6th edition, 2011.

[27] C.-W. Ko, J. Lee, and M. Queyranne. An exact algorithm for maximum entropy sampling. *Operations Research*, 43(4):684–691, 1995.

[28] A. Kuhnle. Interlaced greedy algorithm for maximization of submodular functions in nearly linear time. In *Advances in Neural Information Processing Systems (NeurIPS 2019)*, volume 32. Curran Associates, Inc., 2019.

[29] A. Kuhnle. Nearly linear-time, parallelizable algorithms for non-monotone submodular maximization. In *Proceedings of the 35th AAAI conference on artificial intelligence (AAAI 2021)*, volume 35, pages 8200–8208. AAAI Press, 2021.

[30] A. Kulesza and B. Taskar. $k$-DPPs: Fixed-size determinantal point processes. In *Proceedings of the 28th International Conference on Machine Learning (ICML 2011)*, pages 1193–1200. Omnipress, 2011.

[31] A. Kulesza and B. Taskar. Determinantal point processes for machine learning. *Foundations and Trends® in Machine Learning*, 5(2–3):123–286, 2012.

[32] C. Launay, B. Galerne, and A. Desolneux. Exact sampling of determinantal point processes without eigendecomposition. *Journal of Applied Probability*, 57(4):1198–1221, 2020.

[33] C. Li, S. Sra, and S. Jegelka. Fast mixing Markov chains for strongly Rayleigh measures, DPPs, and constrained sampling. In *Advances in Neural Information Processing Systems (NeurIPS 2016)*, volume 29. Curran Associates, Inc., 2016.

[34] O. Macchi. The coincidence approach to stochastic point processes. *Advances in Applied Probability*, 7(1):83–122, 1975.

[35] S. Mahabadi, P. Indyk, S. O. Gharan, and A. Rezaei. Composable core-sets for determinant maximization: A simple near-optimal algorithm. In *Proceedings of the 36th International Conference on Machine Learning (ICML 2019)*, volume 97, pages 4254–4263. PMLR, 2019.

[36] S. Mahabadi, I. Razenshteyn, D. P. Woodruff, and S. Zhou. Non-adaptive adaptive sampling on turnstile streams. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing (STOC 2020)*, pages 1251–1264. ACM, 2020.

[37] M. Minoux. Accelerated greedy algorithms for maximizing submodular set functions. In *Optimization Techniques*, pages 234–243. Springer Berlin Heidelberg, 1978.

[38] B. Mirzasoleiman, A. Badanidiyuru, A. Karbasi, J. Vondrak, and A. Krause. Lazier than lazy greedy. In *Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI 2015)*, volume 29, pages 1812–1818. AAAI Press, 2015.

[39] T. Nakamura, S. Sakaue, K. Fujii, Y. Harabuchi, S. Maeda, and S. Iwata. Selecting molecules with diverse structures and properties by maximizing submodular functions of descriptors learned with graph neural networks. *Scientific Reports*, 12(1):1124, 2022.

[40] G. L. Nemhauser and L. A. Wolsey. Best algorithms for approximating the maximum of a submodular set function. *Mathematics of Operations Research*, 3(3):177–188, 1978.

[41] G. L. Nemhauser, L. A. Wolsey, and M. L. Fisher. An analysis of approximations for maximizing submodular set functions—I. *Mathematical Programming*, 14(1):265–294, 1978.

[42] N. Ohsaka. Some inapproximability results of MAP inference and exponentiated determinantal point processes. *Journal of Artificial Intelligence Research*, 73:709–735, 2022.

[43] S. Sakaue. Guarantees of stochastic greedy algorithms for non-monotone submodular maximization with cardinality constraints. In *Proceedings of the 23rd International Conference on Artificial Intelligence and Statistics (AISTATS 2020)*, volume 108, pages 11–21. PMLR, 2020.

[44] D. Sharma, A. Kapoor, and A. Deshpande. On greedy maximization of entropy. In *Proceedings of the 32nd International Conference on Machine Learning (ICML 2015)*, volume 37, pages 1330–1338. PMLR, 2015.

## Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes] See Abstract and Section 1.
   (b) Did you describe the limitations of your work? [Yes] The remarkable limitation is that our LAZYFASTGREEDY does not improve the worst-case time complexity, and it is clear from the description in Section 3.1.
   (c) Did you discuss any potential negative societal impacts of your work? [No] This is a purely algorithmic study, and no negative societal impacts are expected.
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [Yes] See Section 2. Additional assumptions required in the unconstrained setting is described in Section 4.
   (b) Did you include complete proofs of all theoretical results? [Yes] All theoretical statements are followed by their proofs.
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] See the supplementary material.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [N/A] No training is performed in our experiments.
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [No] Although some algorithms are randomized, it is obvious that the randomness does not largely affect our experimental results.
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] See Section 5.
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes] See [5, 25].
   (b) Did you mention the license of the assets? [No] We used publicly available standard datasets.
   (c) Did you include any new assets either in the supplemental material or as a URL? [Yes] The codes of the algorithms are provided in the supplementary material.
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [No] Since the datasets are publicly available standard ones, no such discussion seems to be necessary.
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [No] Since the datasets are publicly available standard ones, no such discussion seems to be necessary.
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]

(b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]

(c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

# Appendix

## A Extension to random, stochastic, and interlace greedy algorithms

We consider cardinality-constrained DPP MAP inference and explain how to extend our "lazy + fast" idea to other greedy-type algorithms: RANDOMGREEDY [10], STOCHASTICGREEDY [38, 43], and INTERLACEGREEDY [28]. Roughly speaking, those algorithms have in common the process of finding an element with the largest marginal gain, which we can do efficiently with our "lazy + fast" technique. We also present experimental results on those algorithms.

In the analysis of RANDOMGREEDY [10], STOCHASTICGREEDY [43], and INTERLACEGREEDY [28], $n \geq 2k$, $n \geq 3k$, and $n \geq 4k$, respectively, are assumed. Thus, we below make the same assumptions.

### A.1 Lazy and fast random greedy algorithm

We consider RANDOMGREEDY [10], a $1/\mathrm{e}$-approximation algorithm for non-monotone submodular function maximization with a cardinality constraint. It works as follows. First, we add $2k$ dummy elements that do not affect the function value to the ground set in advance. In each step, RANDOMGREEDY finds a set $M$ of $k$ elements with the top-$k$ marginal gains, draws an element $i$ from $M$ uniformly at random, and then adds $i$ to the solution. This procedure is repeated $k$ times.

For convenience, we consider another equivalent algorithm. In each step of the above original version, an element with the $l$th largest marginal gain is added to a current solution, where $l$ is drawn from $[k]$ uniformly at random. Since there always exist at least $k$ remaining dummy elements, the added element always has a non-negative marginal gain in every step. Considering the above, each step of the original RANDOMGREEDY can be equivalently performed without dummy elements as follows. Let $S \subseteq [n]$ be a current solution. We draw $l$ from the uniform distribution on $[k]$. ($\overline{S}$ has at least $l$ elements since $n \geq 2k$ implies $|\overline{S}| \geq n-k \geq k \geq l$.) If an element with the $l$th largest marginal gain, denoted by $i \in \overline{S}$, satisfies $f_i(S) \geq 0$, we add $i$ to $S$. Otherwise, we add nothing to $S$, corresponding to adding a dummy element in the original algorithm. Note that each step of this algorithm requires finding elements with the top-$l$ marginal gains.

We below explain how to use the "lazy + fast" idea for finding $M$, a set of elements with the top-$l$ marginal gains. First, we compute $d_i = \sqrt{L_{i,i}}$ for every $i \in [n]$ and push them into a priority queue. In each $t$th step, given a current solution $S$, $l \in [k]$, and $M = \emptyset$, we find an element $i$ with the largest $f_i(S)$ among $\overline{S \cup M}$ by calling UPDATEROW (defined in Algorithm 2) and add $i$ to $M$. We repeat this until $|M| = l$ or $d_i < 1$ holds, where $d_i < 1$ implies that the $l$th largest marginal gain is negative. After updating the current solution $S$, we insert back the elements that have not been added into the priority queue. Algorithm 4 presents a formal description of this algorithm.

Let $u = \sum_{i \in [n]} u_i$ be the number of computed off-diagonals of $V$ at the end of the algorithm, which affects the time complexity of Algorithm 4 as follows.

**Theorem A.1.** *Algorithm 4 returns a solution obtained by* RANDOMGREEDY *in* $\mathrm{O}(nd + u(d + \log n))$ *time. If the lazy update works best and worst, it runs in* $\mathrm{O}(nd + k^2(d + \log n))$ *and* $\mathrm{O}(kn(d + \log n))$ *time, respectively.*

*Proof.* The correctness of Algorithm 4 and the time complexity depending on $u$ follow from similar arguments to those in Section 3.1. We below discuss the best and worst cases. For ease of deriving upper bounds on the time complexity, we suppose $l = k$ to hold in every step, which is the worst scenario for the randomness of the algorithm.

If the lazy update works best, in each step, it picks $k-1$ elements that have belonged to $M$ one step before and a new element whose row in a Cholesky factor has not been computed. Then, in each $t$th step ($t = 2, 3, \cdots, k$), $(k-1) + (t-1)$ off-diagonals are computed, hence $u = \sum_{t=2}^{k}((k-1)+(t-1)) = \mathrm{O}(k^2)$. In addition, since $k-1$ elements are inserted in each step, updating the priority queue takes $\mathrm{O}(k^2 \log n)$ in total. Thus, the overall running time is $\mathrm{O}(nd + k^2(d + \log n))$.

We then turn to the worst case. In the $t$th step ($t = 2, 3, \ldots, k$), all the off-diagonals in $V[\overline{S}, j_{t-1}]$ are calculated to determine the top-$k$ element. Hence, it computes as many off-diagonals as FASTGREEDY, i.e., $u = (k-1)(n - k/2)$. Therefore, it takes $\mathrm{O}(kn(d + \log n))$ time. □

**Algorithm 4** Lazy and fast RANDOMGREEDY for DPP MAP inference

$V \leftarrow O$, $\boldsymbol{d} \leftarrow \left(\sqrt{L_{i,i}}\right)_{i\in[n]}$, $\boldsymbol{u} \leftarrow \boldsymbol{0}$, $S \leftarrow \emptyset$
Construct a priority queue of $\boldsymbol{d}$
**for** $t = 1$ to $k$ **do**
  Draw $l$ from the uniform distribution on $[k]$ ▷ $l \leq |\overline{S}|$ always holds since $n \geq 2k$
  $M \leftarrow \emptyset$
  **while** $|M| < l$ **do**
    Take $i \in \arg\max_{i' \in \overline{S \cup M}} d_{i'}$ ▷ Pop the largest one from the priority queue
    UPDATEROW$(V, \boldsymbol{d}, \boldsymbol{u}; i, S, L)$ ▷ $S = \{j_1, j_2, \ldots, j_{|S|}\}$
    **if** $d_i \geq \max_{i' \in \overline{S \cup M}} d_{i'}$ **then** ▷ Otherwise insert $d_i$ into the priority queue
      **if** $d_i < 1$ **then**
        **break** ▷ The $l$th largest marginal gain must be negative
      $M \leftarrow M \cup \{i\}$
      **if** $|M| = l$ **then**
        $j_{|S|+1} \leftarrow i$ ▷ $i$ has the $l$th largest non-negative marginal gain
        $S \leftarrow S \cup \{j_{|S|+1}\}$
  Insert $\{d_i\}_{i \in M \setminus S}$ into the priority queue
**return** $S$

In the above proof, we have assumed the worst scenario of $l = k$, which is pessimistic in practice. We here discuss the optimistic case of $l = 1$ to derive the range of possible $u$ values. In this case, Algorithm 4 selects an element with the largest marginal gain in each step as with LAZYFASTGREEDY. Hence, $u = k(k-1)/2$ off-diagonals of $(PV)[S]$ are computed, which is the lower bound on $u$. The upper bound on $u$ is $(k-1)(n-k/2)$ as shown in the above proof. Therefore, $u$ can take a value in the range of $[k(k-1)/2, (k-1)(n-k/2)]$.

### A.2 Lazy and fast stochastic greedy algorithm

We then discuss STOCHASTICGREEDY [38], which was initially proposed as a simple and fast variant of GREEDY for maximizing a monotone submodular function under a cardinality constraint. A recent study [43] showed that STOCHASTICGREEDY also enjoys a nearly $1/4$-approximation guarantee even for non-monotone submodular functions if $k \ll n$.

STOCHASTICGREEDY has a parameter $\varepsilon \in (0, 1)$ that controls the trade-off between the running time and approximation ratio (see [38, 43] for details). In each step of STOCHASTICGREEDY, given a current solution $S$, it obtains $R \subseteq \overline{S}$ by sampling $s = \left\lceil \frac{n}{k} \log \frac{1}{\varepsilon} \right\rceil$ elements uniformly at random from $\overline{S}$, selects $j_t \in \arg\max_{i' \in R} f_{i'}(S)$, and adds $j_t$ to $S$ if its marginal gain is positive. This is repeated for $t = 1, \ldots, k$. Note that we have $\log \frac{1}{\varepsilon} \leq k$ since $\left\lceil \frac{n}{k} \log \frac{1}{\varepsilon} \right\rceil = s \leq n$ (otherwise we let $R = \overline{S}$).

We can use the idea of "lazy + fast" to speed up the process of finding an element with the largest marginal gain from $R$. To implement the idea, we use an array that maintains the latest $d_i$ values of each element $i \in [n]$. In each step, we construct a priority queue that maintains $(d_i)_{i \in R}$, where $d_i$ values are given by those stored in the array. We then find an element with the largest marginal gain among $R$ by iteratively executing UPDATEROW with the priority queue. We provide a formal description of this algorithm in Algorithm 5.

Letting $u$ denote the number of computed off-diagonals as above, the time complexity of Algorithm 5 is represented as follows.

**Theorem A.2.** *Algorithm 5 returns a solution obtained by* STOCHASTICGREEDY *in* $\mathrm{O}\big(nd + u\big(d + \log\left(\frac{n}{k}\log\frac{1}{\varepsilon}\right)\right)\big)$ *time. If the lazy update works best and worst, it takes* $\mathrm{O}\big((n+k^2)d\big)\big)$ *and* $\mathrm{O}\big(knd + n\log\left(\frac{1}{\varepsilon}\right)\log\left(\frac{n}{k}\log\frac{1}{\varepsilon}\right)\big)$ *time, respectively.*

*Proof.* The correctness of Algorithm 4 follows from a similar argument to that in Section 3.1.

Since we compute at most $n$ diagonals, the time for computing diagonals is $\mathrm{O}(nd)$ in total. The time required for constructing priority queues of $s$ sampled elements is $\mathrm{O}(sk)$ in total, which is bounded by $\mathrm{O}(nd)$ since $s \leq n$ and $k \leq d$.

**Algorithm 5** Lazy and fast StochasticGreedy for DPP MAP inference

$V \leftarrow O$, $\boldsymbol{d} \leftarrow (+\infty)_{i\in[n]}$, $\boldsymbol{u} \leftarrow \mathbf{0}$, $S \leftarrow \emptyset$, $s \leftarrow \lceil \frac{n}{k} \log \frac{1}{\varepsilon} \rceil$ ▷ Initializaiton of $\boldsymbol{d}$ is deferred
**for** $t = 1$ to $k$ **do**
  Get $R$ by sampling $s$ elements from $\overline{S}$ ▷ $R = \overline{S}$ if $|\overline{S}| \leq s$
  Construct a priority queue with $(d_i)_{i\in R}$ ▷ Let $d_i \leftarrow \sqrt{L_{i,i}}$ if $d_i = +\infty$
  **while** true **do**
    Take $i \in \arg\max_{i'\in R} d_{i'}$ ▷ Pop the largest one from the priority queue
    UpdateRow$(V, \boldsymbol{d}, \boldsymbol{u}; i, S, L)$
    **if** $d_i \geq \max_{i'\in R} d_{i'}$ **then** ▷ Otherwise insert $d_i$ into the priority queue
      **break**
  **if** $d_i > 1$ **then**
    $j_{|S|+1} \leftarrow i$
    $S \leftarrow S \cup \{j_{|S|+1}\}$
**return** $S$

In addition, as in the proof of Theorem 3.2, the total computation time caused by UpdateRow is $\mathrm{O}(ud)$. Furthermore, we do at most $\mathrm{O}(u)$ operations on priority queues in total, taking $\mathrm{O}(u \log s)$ time. Thus, the overall time complexity is $\mathrm{O}\big(nd + u\big(d + \log\big(\frac{n}{k} \log \frac{1}{\varepsilon}\big)\big)\big)$.

We below discuss the best and worst cases. As with the proof of Theorem A.1, we assume the worst scenario for the randomness of the algorithm to derive upper bounds on the time complexity.

In the best case, the element on top of the priority queue is added to $S$ in every step. Consequently, $k(k-1)/2$ off-diagonals of $V$ are computed. In addition, updates of priority queues occur $k$ times, which takes $\mathrm{O}(k \log s)$ ($\lesssim \mathrm{O}(nd)$) time in total. Therefore, the best-case time complexity is written as $\mathrm{O}\Big(nd + \frac{k(k-1)}{2} d\Big) = \mathrm{O}\big((n + k^2)d\big)$.

The worst case occurs if $R \subseteq \overline{S}$ sampled in each step minimizes $\sum_{i\in R'} u_i$ among all $R' \subseteq \overline{S}$ with $|R'| = s$, where $u_i$ values are those of the current step, and if the marginal gains of all elements in $R$ are updated. Let $q$ and $r$ be the quotient and the remainder, respectively, of $n$ divided by $s$, i.e., $n = qs + r$ and $0 \leq r < s$. Then, we have

$$u = \sum_{t=k-q+1}^{k} s(t-1) + \left(r - \frac{k-q}{2}\right)(k-q-1) = \left(n - \frac{k}{2}\right)(k-q-1) + \frac{kq}{2} = \mathrm{O}(nk).$$

Moreover, since priority queues are updated at most $sk$ times, the total time taken for operations on priority queues is $\mathrm{O}(sk \log s) \lesssim \mathrm{O}\big(n \log\big(\frac{1}{\varepsilon}\big) \log\big(\frac{n}{k} \log \frac{1}{\varepsilon}\big)\big)$. Thus, the overall time complexity is $\mathrm{O}\big(knd + n \log\big(\frac{1}{\varepsilon}\big) \log\big(\frac{n}{k} \log \frac{1}{\varepsilon}\big)\big)$. □

Note that by using $\log \frac{1}{\varepsilon} \leq k$, we can confirm that the time complexity bounds in Theorem A.2 are at least as good as those of LazyFastGreedy in Theorem 3.2.

### A.3 Lazy and fast interlace greedy algorithm

The idea of "lazy + fast" is also applicable to InterlaceGreedy [28], which is a $1/4$-approximation algorithm for non-monotone submodular maximization with a cardinality constraint. Note that InterlaceGreedy is a deterministic algorithm, unlike RandomGreedy and StochasticGreedy.

InterlaceGreedy sequentially updates two solution sets. Starting from $A = \emptyset$ and $B = \emptyset$, in each step, it adds $j^A \in \arg\max_{i\in\overline{A\cup B}} f_i(A)$ to $A$ and $j^B \in \arg\max_{i\in\overline{(A\cup\{j^A\})\cup B}} f_i(B)$ to $B$, until $|A| = |B| = k$ holds (or the largest marginal gain turns out to be negative). Then, it repeats the same procedure with different initial solutions $C$ and $D$ consisting of the first element added to $A$. Finally, it returns a set with the largest function value among all sets that have appeared as $A$, $B$, $C$, or $D$ at some step during the execution.

The idea of "lazy + fast" can be applied to InterlaceGreedy by using priority queues for $A$, $B$, $C$, and $D$. First, we initialize two priority queues for $A$ and $B$ with $\big(\sqrt{L_{i,i}}\big)_{i\in[n]}$. In each step, by

**Algorithm 6** Lazy and fast INTERLACEGREEDY for DPP MAP inference

$\left(A^{(t)}\right)_{t=0}^{k}, \left(B^{(t)}\right)_{t=0}^{k} \leftarrow$GETINTERLACEDSETS(NIL, $L$)
$\left(C^{(t)}\right)_{t=0}^{k}, \left(D^{(t)}\right)_{t=0}^{k} \leftarrow$GETINTERLACEDSETS($j_1^A, L$) ▷ $A^{(1)} = \{j_1^A\}$
**return** $S \in \arg\max\{\log\det L[S'] \mid S' = X^{(t)}\ (X \in \{A, B, C, D\}, t = 0, \dots, k)\}$

**function** GETINTERLACEDSETS($j_1, L$)
  $V^S \leftarrow O, V^T \leftarrow O, \boldsymbol{d}^S \leftarrow \left(\sqrt{L_{i,i}}\right)_{i\in[n]}, \boldsymbol{d}^T \leftarrow \left(\sqrt{L_{i,i}}\right)_{i\in[n]}, \boldsymbol{u}^S \leftarrow \boldsymbol{0}, \boldsymbol{u}^T \leftarrow \boldsymbol{0}$
  Construct priority queues of $\boldsymbol{d}^S$ and $\boldsymbol{d}^T$
  $S^{(0)} \leftarrow \emptyset, T^{(0)} \leftarrow \emptyset, t_0 \leftarrow |\{j_t\}| + 1$ ▷ $|\{\text{NIL}\}| = 0$
  **if** $j_1 \neq$ NIL **then**
    $j_1^S \leftarrow j_1, j_1^T \leftarrow j_1, S^{(1)} \leftarrow \{j_1^S\}, T^{(1)} \leftarrow \{j_1^T\}$
    Remove $j_1$ from the priority queues of $\boldsymbol{d}^S$ and $\boldsymbol{d}^T$
  **for** $t = t_0$ to $k$ **do**
    $i \leftarrow$ GETARGMAX($V^S, \boldsymbol{d}^S, \boldsymbol{u}^S; S^{(t-1)}, T^{(t-1)}, L$)
    **if** $i \neq$ NIL and $d_i^S \geq 1$ **then**
      $j_t^S \leftarrow i, S^{(t)} \leftarrow S^{(t-1)} \cup \{j_t^S\}$
      Remove $j_t^S$ from the priority queue of $\boldsymbol{d}^T$
    **else**
      $S^{(t)} \leftarrow S^{(t-1)}$
    $i \leftarrow$ GETARGMAX($V^T, \boldsymbol{d}^T, \boldsymbol{u}^T; T^{(t-1)}, S^{(t)}, L$)
    Repeat Lines 13–17 interchanging $S$ and $T$
  **return** $\left(S^{(t)}\right)_{t=0}^{k}, \left(T^{(t)}\right)_{t=0}^{k}$

**function** GETARGMAX($V, \boldsymbol{d}, \boldsymbol{u}; X, Y, L$)
  **if** $\max_{i'\in\overline{X\cup Y}} d_{i'} < 1$ **then**
    **return** NIL
  **while** true **do**
    Pop $i$ with the largest $d_i$ from the priority queue of $\boldsymbol{d}$
    UPDATEROW($V, \boldsymbol{d}, \boldsymbol{u}; i, X, L$)
    **if** $d_i \geq \max_{i'\in\overline{X\cup Y}} d_{i'}$ **then** ▷ Otherwise insert $d_i$ into the priority queue
      **return** $i$

iteratively using UPDATEROW, we can efficiently find an element with the largest marginal gain for $A$ and $B$, as with LAZYFASTGREEDY. Here, since elements already added to $A$ cannot be added to $B$, if $i$ is added to $A$, we remove $i$ from the priority queue for $B$ and vice versa. We thus construct sequences of solutions, $\left(A^{(t)}\right)_{t=0}^{k}$ and $\left(B^{(t)}\right)_{t=0}^{k}$. Then, we obtain $\left(C^{(t)}\right)_{t=1}^{k}$ and $\left(D^{(t)}\right)_{t=1}^{k}$ in the same manner except that $C$ and $D$ are initially set to $A^{(1)}$. We provide a formal description of this algorithm in Algorithm 6.

For each $X \in \{A, B, C, D\}$, let $V^X$ be a Cholesky factor of $L[X^{(k)}]$ (partially) computed in Algorithm 6, $P^X$ be a permutation matrix such that $P^X V^X$ is lower triangular, $u^X$ be the total number of computed off-diagonals in $P^X V^X$, and $u = u^A + u^B + u^C + u^D$. Then, Algorithm 6 enjoys the following guarantee.

**Theorem A.3.** *Algorithm 6 returns a solution obtained by naive* INTERLACEGREEDY *in* $\mathrm{O}(nd+u(d+\log n))$ *time. If the lazy update works best and worst, it runs in* $\mathrm{O}((n+k^2)d)$ *and* $\mathrm{O}(kn(d+\log n))$ *time, respectively.*

*Proof.* The correctness of Algorithm 4 and the time complexity depending on the $u$ value follow from similar arguments to those in Section 3.1. We below discuss the best and worst cases.

In the best case, UPDATEROW is called up to $4k$ times and for $X \in \{A, B, C, D\}$, $u^X = k(k-1)/2$ off-diagonals of $\left(P^X V^X\right)[X^{(k)}]$ are computed. Therefore, $4 \cdot k(k-1)/2 = 2k(k-1) = \mathrm{O}(k^2)$ off-diagonals are computed, taking $\mathrm{O}(k^2 d)$ time in total. Moreover, the priority queues are updated at most $4k$ times, taking $\mathrm{O}(k \log n)$ ($\lesssim \mathrm{O}(nd)$) time in total. Thus, it runs in $\mathrm{O}((n+k^2)d)$ time.

We then discuss the worst case. In the $t$th step ($t = 2, 3, \ldots, k$) of GETINTERLACEDSETS, the off-diagonals in $V^S[\overline{S^{(t-1)} \cup T^{(t-1)}}, j^S_{t-1}]$ and $V^T[\overline{S^{(t)} \cup T^{(t-1)}}, j^T_{t-1}]$ are computed. Therefore, the total number of computed off-diagonals is

$$u = \sum_{t=2}^{k}((n-2t+2)+(n-2t+1)+(n-2t+3)+(n-2t+2)) = 4(k-1)(n-k).$$

Hence, Algorithm 6 runs in $\mathrm{O}(kn(d+\log n))$ time. □

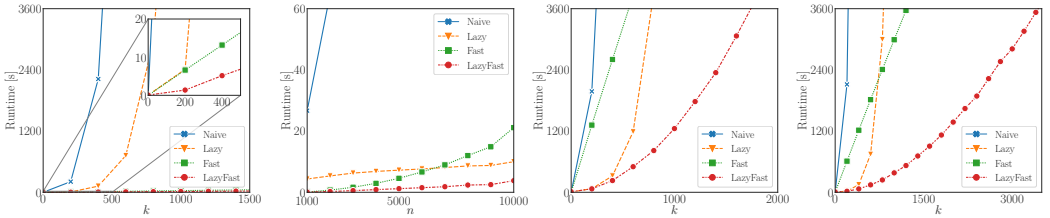
### A.4 Experimental results on random, stochastic, and interlace greedy algorithms

We experimentally compare the running time of the four versions, Naive, Lazy, Fast, and LazyFast, for each of RANDOMGREEDY, STOCHASTICGREEDY, and INTERLACEGREEDY. We also compare objective values achieved by the algorithms.

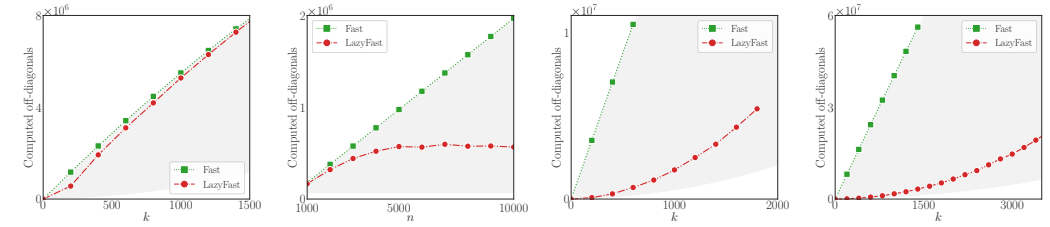
We use the same experimental setups as those in Section 5. We here focus on the $B$-input setting; as we will see below, the results were similar to those of GREEDY, which we confirmed to be true in both $B$- and $L$-input settings. As mentioned above, RANDOMGREEDY, STOCHASTICGREEDY, and INTERLACEGREEDY require $n \geq 2k$, $n \geq 3k$, and $n \geq 4k$, respectively. Therefore, when increasing the $k$ value, we set the upper bound to $\lfloor n/4 \rfloor$ to satisfy all the requirements. Since the procedures of RANDOMGREEDY and STOCHASTICGREEDY depend on the $k$ value as in Algorithms 4 and 5, respectively, we cannot measure their running time incrementally for $k = 1, 2, \ldots, \lfloor n/4 \rfloor$. Therefore, we select evaluation points in increments of 200 and 500 for RANDOMGREEDY and STOCHASTICGREEDY, respectively. We set the parameter $\varepsilon$ of STOCHASTICGREEDY to 0.5.

Figures 4–6 summarize the results of RANDOMGREEDY, STOCHASTICGREEDY, and INTERLACE-GREEDY, respectively. Similar to the results of GREEDY, our LazyFast version was the fastest in all cases. In the synthetic setting with $k = 200$ and real-world settings, our LazyFast ran faster than Fast by avoiding the redundant computation of off-diagonals, while it still ran faster in the synthetic setting with $n = 6000$ due to the cache efficiency discussed in Section 5.1.

Figure 7 summarizes the running times and objective values of the LazyFast version of the algorithms, where ratios relative to those of LAZYFASTGREEDY are shown. Although LAZYFASTGREEDY empirically achieved the best objective values, the other algorithms performed comparably. Note that while the objective values of STOCHASTICGREEDY were the worst, its LazyFast version ran the fastest in many cases. Therefore, when the running time matters more than the solution quality, we can use the LazyFast version of STOCHASTICGREEDY instead of LAZYFASTGREEDY.

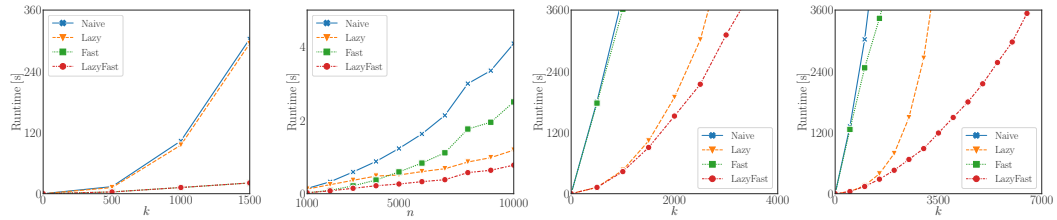



(a) $n = 6000$, Runtime (b) $k = 200$, Runtime (c) Netflix, Runtime (d) MovieLens, Runtime

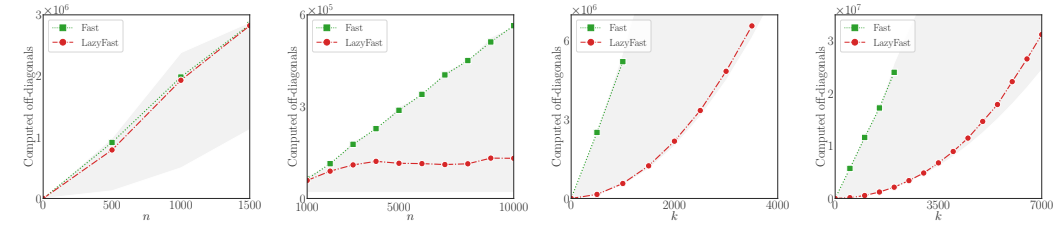



(e) $n = 6000$, Off-diag. (f) $k = 200$, Off-diag. (g) Netflix, Off-diag. (h) MovieLens, Off-diag.

Figure 4: Results of RANDOMGREEDY. In Fig. 4a, enlarged views of lower left parts are shown for visibility. In the lower figures, the gray band indicates the range of the possible number of computed off-diagonals: $[k(k-1)/2, (k-1)(n-k/2)]$.

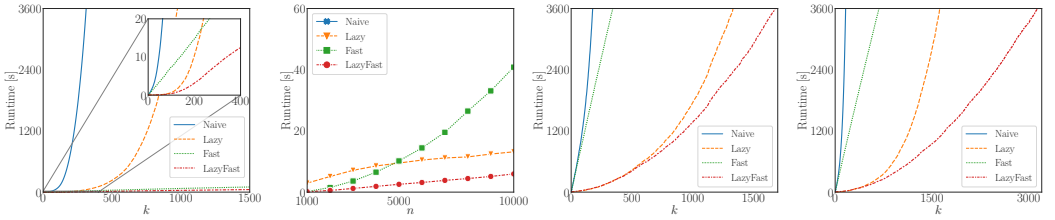



(a) $n = 6000$, Runtime (b) $k = 200$, Runtime (c) Netflix, Runtime (d) MovieLens, Runtime

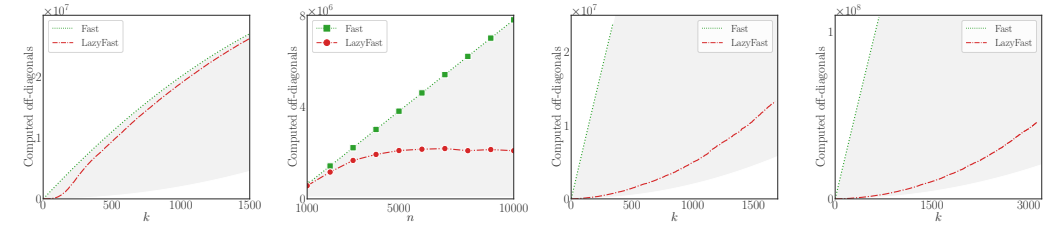



(e) $n = 6000$, Off-diag. (f) $k = 200$, Off-diag. (g) Netflix, Off-diag. (h) MovieLens, Off-diag.

Figure 5: Results of STOCHASTICGREEDY. In the lower figures, the gray band indicates the range of the possible number of computed off-diagonals: $[k(k-1)/2, (n-k/2)(k-q-1)+kq/2]$, where $q \in \mathbb{N}$ is the quotient of $n$ divided by $s$.

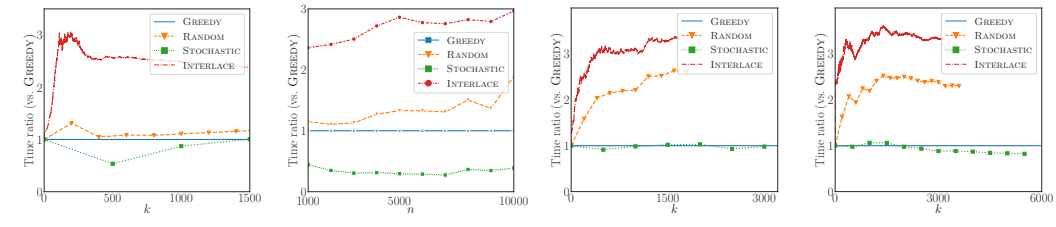



(a) $n = 6000$, Runtime (b) $k = 200$, Runtime (c) Netflix, Runtime (d) MovieLens, Runtime

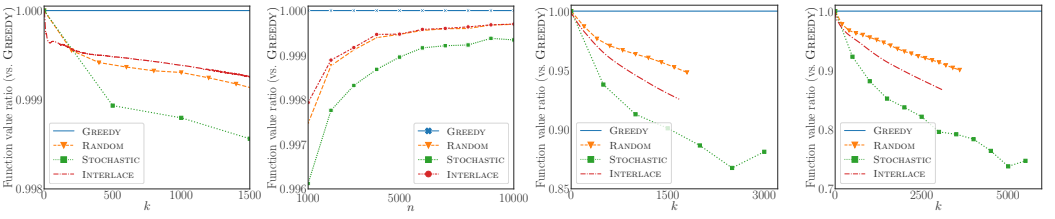



(e) $n = 6000$, Off-diag. (f) $k = 200$, Off-diag. (g) Netflix, Off-diag. (h) MovieLens, Off-diag.

Figure 6: Results of INTERLACEGREEDY. In Fig. 6a, enlarged views of lower left parts are shown for visibility. In Fig. 6b, runtime of Naive is not shown since it did not finish in 60 seconds even for $n = 1000$. In the lower figures, the gray band indicates the range of the possible number of computed off-diagonals: $[2k(k-1), 4(n-k)(k-1)]$.




(a) $n = 6000$, Runtime (b) $k = 200$, Runtime (c) Netflix, Runtime (d) MovieLens, Runtime




(e) $n = 6000$, Obj. value (f) $k = 200$, Obj. value (g) Netflix, Obj. value (h) MovieLens, Obj. value

Figure 7: Running-time and objective-value ratios relative to those of LAZYFASTGREEDY.