Fast Dynamic Sampling for Determinantal Point Processes

Zhao Song
Adobe Research

Junze Yin
Boston University

Lichen Zhang
MIT

Ruizhe Zhang
UC Berkeley

Abstract

In this work, we provide fast dynamic algorithms for repeatedly sampling from distributions characterized by Determinantal Point Processes (DPPs) and Nonsymmetric Determinantal Point Processes (NDPPs). DPPs are a very well-studied class of distributions on subsets of items drawn from a ground set of cardinality n characterized by a symmetric $n \times n$ kernel matrix L such that the probability of any subset is proportional to the determinant of its corresponding principal submatrix. Recent work has shown that the kernel symmetry constraint can be relaxed, leading to NDPPs, which can better model data in several machine learning applications.

Given a low-rank kernel matrix $\mathcal{L} = L + L^\top \in \mathbb{R}^{n \times n}$ and its corresponding eigendecomposition specified by $\{\lambda_i, u_i\}_{i=1}^d$ where $d \leq n$ is the rank, we design a data structure that uses $O(nd)$ space and preprocesses data in $O(nd^{\omega-1})$ time where $\omega \approx 2.37$ is the exponent of matrix multiplication. The data structure can generate a sample according to DPP distribution in time $O(|E|^3 \log n + |E|^{\omega-1} d^2)$ or according to NDPP distribution in time $O((|E|^3 \log n + |E|^{\omega-1} d^2)(1+w)^d)$ for E being the sampled indices and w is a data-dependent parameter. This improves upon the space and preprocessing time over prior works, and achieves a state-of-the-art sampling time when the sampling set is relatively dense. At the heart of our data structure is an efficient sampling tree that can leverage batch initialization and fast inner product query simultaneously.

Proceedings of the 27th International Conference on Artificial Intelligence and Statistics (AISTATS) 2024, Valencia, Spain. PMLR: Volume 238. Copyright 2024 by the author(s).

1 INTRODUCTION

Determinantal point processes (DPPs) are a very well-studied class of distributions on subsets of items. A DPP is characterized by an $n \times n$ symmetric positive semidefinite kernel L where n is the cardinality of the ground set. The probability of sampling any subset is proportional to the determinant of its corresponding principal submatrix in the kernel L. Following their introduction in machine learning by the seminal work of Kulesza and Taskar (2012), DPPs have been adopted widely for several applications such as neural network compression (Mariet and Sra, 2016), text and video summarization (Perez-Beltrachini and Lapata, 2021), and coreset construction (Tremblay et al., 2019). Most recent applications of DPPs focus exclusively on k-DPPs, where a parameter $k \leq n$ is provided, and the distribution is only over the subsets of fixed size k (rather than a random size) such that the probability of any particular subset of size k is proportional to the determinant of its corresponding principal submatrix in L.

Due to the kernel symmetry constraint, DPPs can only model negative correlations between items in the ground set. Recent works (Gartrell et al., 2019, 2021) have shown that relaxing this kernel symmetry constraint can lead to significantly better predictive performance for several machine learning tasks. For example, consider the case of an online marketplace selling electronics: several sets of items have positive correlations like ink cartridges are often bought together with printers. To model such relations, the Nonsymmetric Determinantal Point Processes (NDPPs) are introduced.

Sampling algorithms for DPPs are useful for varied applications of DPPs such as low-rank approximations (Dereziński et al., 2020a), active learning (Biyik et al., 2019), Gaussian processes (Kathuria and Deshpande, 2016), and experimental design (Dereziński et al., 2020b). Although sampling algorithms for symmetric DPPs have been extensively studied (Gillenwater et al., 2019; Dereziński et al., 2019; Poulson, 2020), the study of sampling algorithms for NDPPs is in a very nascent stage. Very recently, the study of sampling algorithms for NDPPs was initiated by Han et al. (2022), in which they provide linear time algorithms for exact sampling from NDPPs with a low-rank decomposition. Additionally, they provide a sub-linear time rejection sampling algorithm for a special class of NDPPs, which they call orthogonal NDPPs. Prior work on DPP sampling has focused extensively on the case where the kernel matrix is *static*. In practical settings, it is often the case that the underlying datasets constantly keep evolving, and so fast *dynamic* sampling algorithms which can adapt to updates of the matrix are quite useful. In the case of symmetric DPPs, there is also a dynamic algorithm for learning low-rank DPPs (Osogami et al., 2018).

In this work, we provide fast dynamic algorithms for repeated sampling from distributions characterized by determinantal point processes (DPPs) and nonsymmetric DPPs (NDPPs). In particular, we will only consider *low-rank* version of DPP (NDPP) (Kulesza and Taskar, 2012; Han et al., 2022).

We first formally define DPP and NDPP, then we give a mathematical definition of the sampling problem we solve.

Definition 1.1 (Determinantal point process (DPP))**.** *Given item set $[n] := \{1, 2, \cdots, n\}$ (see the definition in section 3) and an $n \times n$ matrix L (called the kernel), the determinantal point process (DPP) on $[n]$ is a process to sample a subset $Y \subseteq [n]$ with probability proportional to the determinant of the corresponding principal submatrix L_Y of L, which is formed by taking rows and columns in Y.*

For NDPP, the kernel is a nonsymmetric positive semidefinite matrix defined as follows.

Definition 1.2 (Nonsymmetric positive semidefinite)**.** *A matrix $X \in \mathbb{R}^{n \times n}$ is called nonsymmetric positive semidefinite (nPSD) if $X + X^\top$ is positive semidefinite.*

Definition 1.3 (Nonsymmetric determinantal point process (NDPP))**.** *Given an item set $[n]$ and an $n \times n$ nPSD matrix L, let $\mathcal{L} := L + L^\top$ be the kernel.*

The nonsymmetric determinantal point process (NDPP) on $[n]$ is a process to sample a subset $Y \subseteq [n]$ with probability proportional to the determinant of the corresponding principal submatrix L_Y of L, which is formed by taking rows and columns in Y.

The DPP/NDPP sampling problem is defined as follows:

Definition 1.4 (Sampling Oracle of (N)DPP)**.** *Given the eigendecomposition $\{\lambda_i, u_i\}_{i=1}^d$ of kernel $\mathcal{L}$ where u_i is n-dimensional for all $i \in [d]$, the goal is to design a data structure that supports the following operations:*

- $\textsc{Init}(\{\lambda_i, u_i\}_{i=1}^d, n \in \mathbb{N}_+, d \in \mathbb{N}_+)$. *It takes the eigendecomposition $\{\lambda_i, u_i\}_{i=1}^d$ of the kernel $\mathcal{L}$ as inputs and initializes.*
- $\textsc{Query}()$. *It outputs a set of indices $Y \subset [n]$ following (N)DPP (Definition 1.3) of $\mathcal{L}$.*

The main contribution of this work is a new DPP sampling algorithm that improves the prior approach (Gillenwater et al., 2019) in preprocessing time and space consumption. Moreover, in the dense sample regime (when k is close to d), our result also improves the query time. The key techniques of our algorithm are batched computation of sampling probabilities and efficient low-rank maintenance of DPP matrices. As a consequence, we also obtain a fast NDPP sampling algorithm, improving the result of Han et al. (2022).

1.1 Our Results

We present our main results as follows:

Theorem 1.5 (DPP sampling data structure, informal version of Theorem 5.1)**.** *There exists a data structure (*$\textsc{EfficientPrepDPP}$*) that uses $O(nd)$ space with the following procedures:*

- $\textsc{Init}(\{\lambda_i, u_i\}_{i=1}^d, n \in \mathbb{N}_+, d \in \mathbb{N}_+)$. *Taken the eigendecomposition $\{\lambda_i, u_i\}_{i=1}^d$ of kernel $\mathcal{L}$ as input where $\lambda_i \in \mathbb{R}, u_i \in \mathbb{R}^n$, the data structure initializes in $O(nd^{\omega-1})$ time.*[1]
- $\textsc{Query}()$. *It outputs a list of indices $Y \subset [n]$ with $|Y| = k \leq d$ following NDPP distribution in time $O(k^3 \log n + k^{\omega-1} d^2)$.*

Remark 1.6. *Compared with Gillenwater et al. (2019), which gave an DPP algorithm with $O(nd^2)$ preprocessing time and $k^4 + d$ query time using $O(nd^2)$ space, our result improves the preprocessing time and the space complexity. Furthermore, when the sample set is dense (i.e., $k = \Omega(d^{0.75})$), our algorithm also has a faster query procedure.*

As a corollary, we also get the following NDPP sampling data structure, where the proof is provided in Appendix B.

Corollary 1.7 (NDPP sampling data structure, informal version of Corollary B.1)**.** *There exists a data structure that uses $O(nd)$ space with the following procedures:*

- $\textsc{Init}(\{\lambda_i, u_i\}_{i=1}^d, n \in \mathbb{N}_+, d \in \mathbb{N}_+)$. *Taken the eigendecomposition $\{\lambda_i, u_i\}_{i=1}^d$ of kernel $\mathcal{L}$ (Definition 1.3) as input where $\lambda_i \in \mathbb{R}, u_i \in \mathbb{R}^n$, the data structure initializes in $O(nd^{\omega-1})$ time.*

[1]We use ω to denote the exponent of matrix multiplication. For more details, we refer the readers to Fact 3.1.

- QUERY(). *It outputs a list of indices $Y \subset [n]$ with $|Y| = k \leq d$ following NDPP distribution in time $O((k^3 \log n + k^{\omega-1} d^2)(1+w)^d)$, where w is a data-dependent parameter*[2].

Roadmap. In Section 2, we discuss prior work on DPPs and NDPPs. In Section 3, we provide some preliminary notations and tools. In Section 4, we introduce our techniques of fast (N)DPP sampling algorithms. In Section 5, we show our DPP sampling data structure and prove the running time and correctness. In Section 6, we draw the conclusion.

2 RELATED WORK

Symmetric DPP. DPP sampling is an extensively studied topic with several applications in machine learning (Li et al., 2016b; Celis et al., 2018). Initial work for exact sampling from k-DPPs (Kulesza and Taskar, 2011) relied on an expensive eigendecomposition computation of the kernel matrix. Recently, there has been a lot of progress in the fast exact sampling of k-DPPs by avoiding the eigendecomposition computation of the complete kernel matrix (Calandriello et al., 2020; Gillenwater et al., 2019). Due to their use in multiple varied applications, there has also been a lot of work for *approximate sampling* for DPPs (Anari et al., 2016; Li et al., 2016c,a). Most approximate samplers for DPPs rely on MCMC methods.

Nonsymmetric DPP. The first class of Nonsymmetric DPPs (NDPPs) to be studied were *signed* DPPs (Brunel, 2018). Following this, Gartrell et al. (2019) showed that a more general class of NDPPs, where the kernel matrix L was a P_0-matrix could be learned in $O(n^3)$ where n is the size of the ground set. Gartrell et al. (2021) improved this by providing an $O(n)$ time learning algorithm for NDPPs. Note that all the prior mentioned papers focus only on the learning and inference problems for NDPPs. Sampling algorithms were not discussed.

3 PRELIMINARY

Here, we first explain the notations that we will use in this paper. Then, we present some basic definitions and properties from linear algebra in order to support our analysis.

Notations. We use $\mathbb{N}_+$ to denote all the positive integers. For a matrix A, we use A^{-1} to denote its inverse. We use $A^\top$ to denote the transpose of matrix A. We use $\det(A)$ to denote the determinant of a matrix. We use $\mathbb{E}[\cdot]$ for expectation. We use $\Pr[\cdot]$ for probability. We use $\propto$ to denote proportional to. For any function f, we use $\widetilde{O}(f)$ to denote $f \cdot \mathrm{poly}(\log f)$. For any positive integer n, we use $[n]$ to denote $\{1, 2, \cdots, n\}$. For two vectors u and v, we use $\langle u, v\rangle$ to denote their inner product. For two matrices A and B, we also use $\langle A, B\rangle$ to denote their inner product.

For a matrix A, we use A_{ij} to denote its entry in the i-th row and j-th column. We use $A_{j,:}$ to denote its j-th row. For a set E, we use A_E to denote the submatrix of A formed by taking rows and columns E. We say a matrix P is a projection matrix if $P = P^2$.

Fact 3.1 (Fast Matrix Multiplication)**.** *Given a size $a \times b$ matrix and a size $b \times c$ matrix, we use $\mathcal{T}_{\mathrm{mat}}(a, b, c)$ to denote the time of multiplying two matrices together. We use ω to denote the exponent of matrix multiplication which means $\mathcal{T}_{\mathrm{mat}}(n, n, n) = n^{\omega+o(1)}$. Currently, $\omega \approx 2.373$ (Williams, 2012; Le Gall, 2014; Alman and Williams, 2021; Duan et al., 2023; Gall, 2024; Williams et al., 2024).*

Next, we state some basic definitions and tools in linear algebra.

Fact 3.2 (Matrix Woodbury Identity, (Woodbury, 1949, 1950))**.** *Given four matrices A is $n \times n$, C is $k \times k$, U is $n \times k$, and V is $k \times n$. The Woodbury matrix identity is*

$$\begin{aligned}&(A + UCV)^{-1}\\ =\ &A^{-1} - A^{-1}U \cdot (C^{-1} + VA^{-1}U)^{-1} VA^{-1},\end{aligned}$$

This can be derived using blockwise matrix inversion.

Fact 3.3 (Sherman-Morrison Formula)**.** *Suppose that $A \in \mathbb{R}^{n\times n}$ is an invertible square matrix and $u, v \in \mathbb{R}^n$ are column vectors.*

Then, the $n \times n$ matrix $A + uv^\top$ is invertible if and only if $1 + v^\top A^{-1} u \neq 0$.

In this case, $(A + uv^\top)^{-1} = A^{-1} - \frac{A^{-1}uv^\top A^{-1}}{1+v^\top A^{-1}u}$.

4 TECHNIQUE OVERVIEW

In Section 4.1, we first introduce our techniques for efficiently sampling DPPs. Then, in Section 4.2, we show how to obtain an NDPP sampling algorithm from a DPP algorithm. Meanwhile, when presenting our techniques, we also show the relationship between our work and prior work.

4.1 Techniques for DPP Sampling

From determinant to inner product Before designing any data structure, we first need to understand

[2]As observed in Han et al. (2022), in many practical datasets, the factor $(1+w)^d \ll n$.

our goal. To compute the sample probability for a subset $Y \subset [n]$ with respect to kernel $\mathcal{L}$, we utilize the equation

$$\begin{aligned} & \Pr_{\mathcal{L}}[Y] \\ = & \frac{\det(\mathcal{L}_Y)}{\det(\mathcal{L}+I)} \\ = & \sum_{E \subseteq [d], |E|=|Y|} \det(U_{Y,E} U_{Y,E}^\top) \cdot \prod_{i \in E} \frac{\lambda_i}{\lambda_i+1} \prod_{i \notin E} \frac{1}{\lambda_i+1}. \end{aligned} \tag{1}$$

Fix E, consider the distribution of Y

$$\Pr_{E,\mathcal{L}}[Y] = \det(U_{Y,E} U_{Y,E}^\top) \cdot \prod_{i \in E} \frac{\lambda_i}{\lambda_i+1} \prod_{i \notin E} \frac{1}{\lambda_i+1}. \tag{2}$$

Notice that $U_{Y,E} U_{Y,E}^\top$ is the principal submatrix of $U_{:,E} U_{:,E}^\top$ corresponding to Y, sampling Y following Eq. (2) is equivalent to executing DPP with kernel $U_{:,E} U_{:,E}^\top$. Thus, the DPP sampling process can be divided into 2 steps:

1. randomly select a set E with $|E| = |Y|$ and with probability corresponding to
$$\prod_{i \in E} \frac{\lambda_i}{\lambda_i+1} \prod_{i \notin E} \frac{1}{\lambda_i+1},$$
which can be easily achieved by d coin tossings;
2. then, execute DPP with kernel $U_{:,E} U_{:,E}^\top$.

Step 2 can also be accelerated via a binary tree structure. Given DPP kernel $M := U_{:,E} U_{:,E}^\top$, initialize a set Y as an empty set, to satisfy DPP probability distribution, the algorithm adds elements j into Y repeatedly, according to the following probability distribution:

$$\Pr[j \mid Y] = K_{j,j} - K_{j,Y} (K_Y)^{-1} K_{Y,j}.$$

Note that if we define

$$Q^Y := I_{|E|} - U_{Y,E}^\top (U_{Y,E} U_{Y,E}^\top)^{-1} U_{Y,E}$$

then

$$\begin{aligned} K_{j,j} - K_{j,Y}(K_Y)^{-1} K_{Y,j} &= U_{j,E} Q^Y U_{j,E}^\top \\ &= \langle Q^Y, (U_{j,:}^\top U_{j,:})_E \rangle, \end{aligned}$$

where the first step follows from the definition of Q^Y and the second step follows from the definition of the inner product.

Thus, to compute the correct conditional probability $\Pr[j|Y]$, it suffices to estimate the inner product between a "query matrix" Q^Y and an outer product matrix that is determined by the subset of rows of input data.

Therefore, our goal is to design a data structure that can

- Preprocess the matrix U quickly.
- Support the query in the form of computing the inner product between a query matrix Q and a subset of preprocessed rows efficiently.

Batched Probability Computation. We first observe that all prior works process the inner product one by one, so to build up the tree of depth $\log n$, it requires paying $O(nd^2)$ time to construct all leaf nodes, then propagate the sum to the root in $O(nd^2)$ time. In contrast, our approach leverages the power of *batching*: instead of using a tree whose leaf corresponds to *one* vector, we design the tree so that its leaf corresponds to d vectors. Specifically, we partition the vectors into n/d blocks, each block contains d vectors. For preprocessing, we compute n/d different matrix-matrix multiplication of size $d \times d$, which takes $O(nd^{\omega-1})$ time in total. With fast matrix multiplication, we improve upon prior best $O(nd^2)$ time (Gillenwater et al., 2019). During the query, we need to use a more sophisticated decoding mechanism by computing matrix-matrix multiplication of $d \times k$ and $k \times d$. Compared to the state-of-the-art (Gillenwater et al., 2019; Han et al., 2022) which can have a potentially exponential dependence on d, our algorithm improves upon their preprocessing time, dependence on k, and space consumption.

Low-Rank Change of Projection and Maintenance. At each iteration, we need to form a new projection matrix

$$B_t = Z_t^\top (Z_t Z_t^\top)^{-1} Z_t$$

where Z_t has only been inserted one new row compared to Z_{t-1}. Given the projection from the last iteration B_{t-1}, we can use matrix Woodbury identity to update the inverse much faster than directly forming B_t.

4.2 Techniques for NDPP Sampling

In order to sample from an NDPP distribution in sublinear time, Han et al. (2022) shows that they can use the idea of *rejection sampling* (Von Neumann, 1951). We will first give a high-level overview of how rejection sampling algorithms work and then summarize the key steps of the rejection sampling algorithm of Han et al. (2022). In particular, our goal will be to sample exactly from an NDPP distribution characterized by a kernel $\mathcal{L}$, which we will refer to as the *target distribution*. To sample from any target distribution $f(x)$, any rejection sampling based algorithm needs a proposal distribution $g(x)$ which has the same support as $f(x)$ and which further satisfies the property that $f(x) \leq Mg(x)$ for all x with $M > 1$ being a constant. Proposal distributions need to be carefully chosen such that they satisfy the above property and also it should be known how to sample exactly from the proposal distribution. To simulate sampling from the target distribution, a sample x is drawn from the proposal distribution g and it is then accepted with probability $f(x)/(Mg(x))$. Samples are kept being drawn from g such that at least one sample is picked. Note that for any particular x, we know how to compute $f(x)$ and $g(x)$. Further, M needs to be known in advance. In order to sample exactly from an NDPP distribution with kernel $\mathcal{L}$, Han et al. (2022) showed that we can use a DPP distribution defined by a particular *symmetric* kernel $\widehat{\mathcal{L}}$ which depends on $\mathcal{L}$. Hence, to implement an efficient sampler for NDPP, it is crucial to use a fast sampler for DPP, and we can use our data structure to realize this step.

5 FAST DPP SAMPLING DATA STRUCTURE

In this section, we show our first result: a DPP sampling data structure with fast initialization time. In Section 5.1, we state the main theorem and the pseudocodes of the algorithm. There are three parts to the Algorithm: initialization, query, and sample distribution. In Section 5.4, we prove the running time of initialization; in Section 5.5, we prove the running time of the query; in Section 5.6, we prove the running time of the sample distribution. In Section 5.2 we prove the correctness of the sample. In Section 5.3, we prove the correctness of the query. These two compose the correctness of the algorithm.

5.1 Main Theorem and Algorithm

The main theorem of this section is stated as follows:

Theorem 5.1 (Formal Version of Theorem 1.5). *Let $E \subset [d]$.*

Then, there exists a data structure using $O(nd)$ space with the following procedures:

- $\textsc{Init}(\{\lambda_i, u_i\}_{i=1}^d, n \in \mathbb{N}_+, d \in \mathbb{N}_+)$. *Taken the eigendecomposition $\{\lambda_i, u_i\}_{i=1}^d$ of kernel $\mathcal{L}$ as input, the data structure initializes in $O(nd^{\omega-1})$ time.*

- $\textsc{Query}()$. *It outputs a list of indices $Y \subset [n]$ from NDPP in time*
$$O(|E|^3 \log n + |E|^{\omega-1} d^2).$$

- $\textsc{Sample}(Q \in \mathbb{R}^{|E|\times|E|}, E)$. *Taken a subset E of $[d]$ and a $|E| \times |E|$ matrix Q as input, the data structure samples a $j \in [n]$ with probability proportional to $\langle (u_j u_j^\top)_E, Q\rangle$ in*
$$O(|E|^2 \log n + |E|^{\omega-2} d^2)$$
time, where u_j is the j-th row of U.

Proof. Algorithms 1, 2 and 3 give the desired data structure. The running time follows from Lemma 5.4 (for $\textsc{Init}$), Lemma 5.5 (for $\textsc{Query}$), and Lemma 5.7 (for $\textsc{Sample}$). The sample correctness follows from Lemma 5.2 and Lemma 5.3. □

Algorithm 1 EfficientPrepDPP: Initialization

```
1:  data structure EfficientPrepDPP
2:                                                  ▷ Theorem 5.1
3:  members
4:      Binary tree $T$
5:      Matrix $U \in \mathbb{R}^{n\times d}$
6:      Vectors $u_1, u_2, \cdots, u_d \in \mathbb{R}^n$
7:      Scalars $\lambda_1, \lambda_2, \cdots, \lambda_d \in \mathbb{R}$
8:  end members
9:
10: procedure BuildTree($A \subset [n]$)
11:     if $|A| = d$ then
12:         Write $A$ as $\{j_1, j_2, \cdots, j_d\}$
13:         $V \leftarrow [\, U_{j_1,:}^\top \; U_{j_2,:}^\top \; \cdots \; U_{j_d,:}^\top \,]^\top$
14:         $\mathcal{T}$.index $\leftarrow A$
15:         $\mathcal{T}.V \leftarrow V$
16:         $\mathcal{T}.\Sigma \leftarrow VV^\top$
17:         return $\mathcal{T}$
18:     end if
19:     $A_\ell, A_r \leftarrow$ Split $A$ in half such that $|A_\ell|$ is a multiple of $d$
20:     $\mathcal{T}$.left $\leftarrow$ BuildTree($A_\ell$)
21:     $\mathcal{T}$.right $\leftarrow$ BuildTree($A_r$)
22:     $\mathcal{T}.\Sigma \leftarrow \mathcal{T}$.left.$\Sigma + \mathcal{T}$.right.$\Sigma$
23:     return $\mathcal{T}$
24: end procedure
25:
26: procedure Init($\{\lambda_i, u_i\}_{i=1}^d$)              ▷ Lemma 5.4
27:     $U \leftarrow [\, u_1 \; u_2 \; \cdots \; u_d \,]$
28:     for $i = 1$ to $n$ do
29:         $u_i \leftarrow u_i$
30:     end for
31:     $\mathcal{T}$ = BuildTree($[n]$)
32:     $T$.root $\leftarrow \mathcal{T}$
33: end procedure
34: end data structure
```

Algorithm 2 EfficientPrepDPP: Query

```
1:  data structure EfficientPrepDPP
2:                                              ▷ Theorem 5.1
3:  procedure Query                             ▷ Lemma 5.5
4:      $E \leftarrow \emptyset$
5:      for $i = 1 \to d$ do      ▷ This implies that $|E| \leq d$
6:          $E \leftarrow E \cup \{i\}$ with probability $\lambda_i/(\lambda_i+1)$
7:      end for
8:      $Q \leftarrow \{0\}^{|E|\times|E|}$
9:      $Y \leftarrow \emptyset$
10:     for $t = 1 \to |E|$ do
11:         $j \leftarrow$ Sample$(Q, E)$
12:         $Y \leftarrow Y \cup \{j\}$
13:         $Z_t \leftarrow Z_{t-1} + U_{j,E}$
14:         $B_t \leftarrow Z_{Y,E}^\top (Z_{Y,E} Z_{Y,E}^\top)^{-1} Z_{Y,E}$
15:         $Q \leftarrow I - B_t$
16:     end for
17:     return $Y$
18: end procedure
19: end data structure
```

5.2 Correctness of Sample

The purpose of this section is to prove Lemma 5.2, showing the correctness of the Sample procedure. It immediately implies the correctness of DPP sampling, where the proof is deferred to Lemma 5.3.

Lemma 5.2 (Correctness of Sample)**.** *Procedure* Sample(Q, E) *in Algorithm 2 takes* $Q \in \mathbb{R}^{|E|\times|E|}$ *and* $E \subseteq [d]$ *as inputs and correctly samples an index* $j \in [n]$ *with probability proportional to* $\langle (u_j u_j^\top)_E, Q\rangle$.

Proof. For each node $\mathcal{T}$ of T, define $S(\mathcal{T})$ by:

- if $\mathcal{T}$ is a leaf, then $S(\mathcal{T}) := \mathcal{T}$.index, a d-subset of $[n]$;
- if $\mathcal{T}$ is not a leaf, then $S(\mathcal{T}) := S(\mathcal{T}.\texttt{left}) \cup S(\mathcal{T}.\texttt{left})$.

In addition, for every node $\mathcal{T} \in T$, define event $\mathcal{E}_\mathcal{T}$ to be: Sample$(\mathcal{T}, Q, E)$ being called in the running of Sample$(T.\text{root}, Q, E)$.

We first prove by induction that for every $\mathcal{T} \in T$,

$$\Pr[\mathcal{E}_\mathcal{T}] = \frac{\sum_{j\in S(\mathcal{T})} \langle (u_j u_j^\top)_E, Q\rangle}{\sum_{j\in[n]} \langle (u_j u_j^\top)_E, Q\rangle}. \quad (3)$$

First, obviously it holds for $\mathcal{T} = T$.root, since $S(T.\text{root}) = [n]$.

Then, suppose it holds for some node $\mathcal{T}$, and consider its child nodes. We have

$$\begin{aligned}
&\Pr[\mathcal{E}_{\mathcal{T}.\texttt{left}}] \\
&= \frac{p_\ell}{p_\ell + p_r} \Pr[\mathcal{E}_\mathcal{T}] \\
&= \frac{p_\ell}{p_\ell + p_r} \cdot \frac{\sum_{j\in S(\mathcal{T})} \langle (u_j u_j^\top)_E, Q\rangle}{\sum_{j\in[n]} \langle (u_j u_j^\top)_E, Q\rangle} \\
&= \frac{\sum_{j\in S(\mathcal{T}.\texttt{left})} \langle (u_j u_j^\top)_E, Q\rangle}{\sum_{j\in S(\mathcal{T})} \langle (u_j u_j^\top)_E, Q\rangle} \cdot \frac{\sum_{j\in S(\mathcal{T})} \langle (u_j u_j^\top)_E, Q\rangle}{\sum_{j\in[n]} \langle (u_j u_j^\top)_E, Q\rangle} \\
&= \frac{\sum_{j\in S(\mathcal{T}.\texttt{left})} \langle (u_j u_j^\top)_E, Q\rangle}{\sum_{j\in[n]} \langle (u_j u_j^\top)_E, Q\rangle},
\end{aligned}$$

where the first step follows from Line 28 in Algorithm 3, the second step follows from the induction hypothesis, the third step follows from the definitions of p_ℓ (at Line 25) and p_r (at Line 26), and the last step follows from the direct computation.

Algorithm 3 EfficientPrepDPP: Sample

```
1:  data structure EfficientPrepDPP
2:                                              ▷ Theorem 5.1
3:  procedure Sample_Auxiliary$(\mathcal{T}, Q, E)$
4:                                              ▷ Lemma 5.8
5:      Assert $\mathcal{T}$.index has exactly $d$ elements
6:      Write $\mathcal{T}$.index as $\{j_1, j_2, \cdots, j_d\}$
7:      $V \leftarrow \mathcal{T}.V$
8:      $M \leftarrow V_{E,:}^\top \cdot Q \cdot V_{E,:}$
9:      for $i = 1$ to $d$ do
10:         $p_i \leftarrow M_{ii}$
11:     end for
12:     $u \leftarrow$ uniform$(0, 1)$
13:     for $i = 1$ to $d$ do
14:         if $u \leq (p_1 + \cdots + p_i)/(p_1 + \cdots + p_d)$ then
15:             return $j_i$
16:         end if
17:     end for
18: end procedure
19:
20: procedure Sample$(\mathcal{T} = T.\text{root}, Q, E)$
21:                                             ▷ Lemma 5.7
22:     if $\mathcal{T}$ is a leaf then
23:         return Sample_Auxiliary$(\mathcal{T}, Q, E)$
24:     end if
25:     $p_\ell \leftarrow \langle \mathcal{T}.\text{left}.\Sigma_E, Q\rangle$
26:     $p_r \leftarrow \langle \mathcal{T}.\text{right}.\Sigma_E, Q\rangle$
27:     $u \leftarrow$ uniform$(0, 1)$
28:     if $u \leq \frac{p_\ell}{p_\ell + p_r}$ then
29:         Sample$(\mathcal{T}.\text{left}, Q, E)$
30:     else
31:         Sample$(\mathcal{T}.\text{right}, Q, E)$
32:     end if
33: end procedure
34: end data structure
```

Similarly, for the right-child node, we have

$$\Pr[\mathcal{E}_{\mathcal{T}.\texttt{right}}] = \frac{\sum_{j \in S(\mathcal{T}.\texttt{right})} \langle (u_j u_j^\top)_E, Q\rangle}{\sum_{j\in[n]} \langle (u_j u_j^\top)_E, Q\rangle}.$$

Therefore, by induction, Eq. (3) holds for every node $\mathcal{T} \in T$. In particular, for a leaf $\mathcal{T}$ with $\mathcal{T}$.index $= \{j_1, j_2, \cdots, j_d\}$, we have

$$\Pr[\mathcal{E}_{\mathcal{T}}] = \frac{\sum_{j \in \{j_1, j_2, \cdots, j_d\}} \langle (u_j u_j^\top)_E, Q\rangle}{\sum_{j\in[n]} \langle (u_j u_j^\top)_E, Q\rangle}.$$

Then, the SAMPLE_AUXILIARY procedure is called to sample an index from $\{j_1, j_2, \cdots, j_d\}$. In Line 14, we know that for every $i \in [d]$,

$$\Pr[j_i \text{ is sampled} \mid \mathcal{E}_{\mathcal{T}}] = \frac{\langle (u_{j_i} u_{j_i}^\top)_E, Q\rangle}{\sum_{j \in \{j_1, j_2, \cdots, j_d\}} \langle (u_j u_j^\top)_E, Q\rangle}.$$

Thus, we get that

$$\begin{aligned}\Pr[j_i \text{ is sampled}] = &\ \Pr[\mathcal{E}_{\mathcal{T}}] \cdot \Pr[j_i \text{ is sampled} \mid \mathcal{E}_{\mathcal{T}}] \\ = &\ \frac{\langle (u_{j_i} u_{j_i}^\top)_E, Q\rangle}{\sum_{j\in[n]} \langle (u_j u_j^\top)_E, Q\rangle},\end{aligned}$$

which finishes the proof of the lemma. □

5.3 Correctness of Query

In this section, we provide the correctness proof of our QUERY procedure. This lemma proves that procedure QUERY of data structure EFFICIENTPREPDPP samples a set following exactly from the DPP probability distribution.

Lemma 5.3 (Correctness of QUERY)**.** *The output $Y \subset [n]$ of the* QUERY *procedure in Algorithm 2 satisfies the DPP with kernel $\mathcal{L}$, i.e.,*

$$\Pr[Y] \propto \det(\mathcal{L}_Y).$$

Proof. Suppose the eigendecomposition of $\mathcal{L}$ is $\{(\lambda_i, u_i)\}_{i=1}^d$.

Suppose

$$U := [u_1, \cdots, u_d] \in \mathbb{R}^{n\times d}.$$

Notice that Eq. (1) has shown that

$$\begin{aligned}&\Pr_{\mathcal{L}}[Y] \\ = &\sum_{E\subseteq[d], |E|=|Y|} \det(Z_{Y,E} Z_{Y,E}^\top) \prod_{i\in E} \frac{\lambda_i}{\lambda_i+1} \prod_{i\notin E} \frac{1}{\lambda_i+1} \\ = &\Pr_K[Y] \cdot \prod_{i\in E} \frac{\lambda_i}{\lambda_i+1} \prod_{i\notin E} \frac{1}{\lambda_i+1},\end{aligned}$$

where $M := U_{:,E} U_{:,E}^\top$ is the DPP kernel. Therefore, sampling Y from DPP with kernel $\mathcal{L}$ can be achieved by 2 steps: first, sample a subset E; then, sample a set Y from DPP with kernel M.

In Line 6, we sample each element $i \in [d]$ independently from the Bernoulli distribution with probability $\frac{\lambda_i}{\lambda_i+1}$. Hence, the set E is sampled correctly.

Next, we sample $|E|$ elements in $[n]$ and add them to the output set Y one-by-one in Lines 10 - 16. Let Y be the current set with t elements, where $0 \le t < |E|$. Consider the marginal distribution of the $(t+1)$-th sample element. By Eq. (3) in Gillenwater et al. (2019), we know that for any $j \in [n]$,

$$\begin{aligned}\Pr[j \text{ is sampled} \mid Y] &= K_{j,j} - K_{j,Y}(K_Y)^{-1} K_{Y,j} \\ &= \langle Q_t, U_{E,j} U_{E,j}^\top\rangle,\end{aligned}$$

where

$$Q_t = I - U_{Y,E}^\top (U_{Y,E} U_{Y,E}^\top)^{-1} U_{Y,E}. \tag{4}$$

By the definition of U, we have

$$U_{E,j} U_{E,j}^\top = (u_j u_j^\top)_E.$$

Hence, the marginal distribution of j is the same as the output distribution of the SAMPLE(Q, E) procedure in Algorithm 3.

In Line 15, we know that Q is correctly computed in each iteration according to Eq. (4). And in Line 11, we know that an element j is sampled from the marginal distribution using the SAMPLE procedure.

Therefore, the QUERY procedure can sample a subset $Y \subset [n]$ from the DPP with kernel $\mathcal{L}$. The lemma is then proved. □

5.4 Running Time of Initialization

The following lemma shows the running time of the INIT procedure.

Lemma 5.4. *Procedure* INIT *in Algorithm 1 takes the eigendecomposition $\{\lambda_i, u_i\}_{i=1}^d$ of $\mathcal{L}$ as input and runs in $O(nd^{\omega-1})$ time.*

Proof. The running time of BUILDTREE(A) only depends on $|A|$.

Therefore, we let $f(k)$ denote the running time of BUILDTREE(A) where $|A| = k$ is a multiple of d.

For $k = d$, consider an A such that $|A| = d$, since V times $V^\top$ takes $O(d^\omega)$ time, BUILDTREE(A) times $O(d^\omega)$ time, we have

$$f(d) = O(d^\omega).$$

For $k > d$, consider an A such that $|A| = md$ where $2^t < m \leq 2^{t+1}$, since $\textsc{BuildTree}(A)$ recursively calls $\textsc{BuildTree}(A_\ell)$ and $\textsc{BuildTree}(A_r)$ and spends extra $O(d^2)$ time summing up $\mathcal{T}.\texttt{left}.\Sigma$ and $\mathcal{T}.\texttt{right}.\Sigma$, we have

$$f(md) = f(\lfloor m/2 \rfloor d) + f(\lceil m/2 \rceil d) + cd^2$$

where $c > 0$ is a constant. Hence, we get the following recurrence relations:

$$\begin{cases} f(md) = f(\lfloor m/2 \rfloor d) + f(\lceil m/2 \rceil d) + cd^2 \\ f(d) = O(d^\omega) \end{cases} .$$

Solving the recurrence, we get that Procedure $\textsc{Init}$ in Algorithm 1 takes $f(n) = O(nd^{\omega-1})$ time. □

5.5 Running Time of Query

The purpose of this section is to prove Lemma 5.5, which shows the running time of the $\textsc{Query}$ procedure.

Lemma 5.5 (Running Time of $\textsc{Query}$). *Procedure* $\textsc{Query}$ *outputs a set $Y \subset [n]$ and runs in*

$$O(|E|^3 \log n + |E|^{\omega-1} d^2) = O(k^3 \log n + k^{\omega-1} d^2)$$

time. Here ω is the exponent of matrix multiplication.

Proof. Computing $E \subset [d]$, $Y \subset [n]$ and Q takes $|E|^2$ time. The for-loop from line 5 takes $O(d)$ time.

For the for-loop from line 10, in each iteration,

- computing j takes $O(k^2 \log n + k^{\omega-2} d^2)$ time by Lemma 5.7.
- computing B_t takes $O(k^2)$ time by Lemma 5.6.
- computing Q takes $O(k^2)$ time.

Therefore, the total running time of $\textsc{Query}$ is:

$$\begin{aligned} &\#\text{iterations} \times \text{ cost per iteration} \\ = &\; k \cdot O(k^2 \log n + k^{\omega-2} d^2). \end{aligned}$$

Thus, we complete the proof. □

The following Lemma 5.6 analyzes the running time of a certain part of Algorithm 2, which is used in Lemma 5.5.

Lemma 5.6. *For the for-loop from line 10 to 16 in Algorithm 2, computing $k \times k$ matrix B_t takes $O(k^2)$ time in each iteration.*

Proof. Since Z_t is a rank-1 update of Z_{t-1}, by Claim A.3, $Z_t Z_t^\top$ is a rank-2 update of $Z_{t-1} Z_{t-1}^\top$ and can be computed in $O(k^2)$ time by first computing 3 outer products in $O(k^2)$ time then add them to the prior $Z_{t-1} Z_{t-1}^\top$. By Claim A.4, $G_t := (Z_t Z_t^\top)^{-1} \in \mathbb{R}^{t \times t}$ is a rank-2 update of $(Z_{t-1} Z_{t-1}^\top)^{-1}$, and

$$\begin{aligned} &(Z_t Z_t^\top)^{-1} \\ = &\,(Z_{t-1} Z_{t-1}^\top + UV^\top)^{-1} \\ = &\,(Z_{t-1} Z_{t-1}^\top)^{-1} + (Z_{t-1} Z_{t-1}^\top)^{-1} U \\ &\cdot (I - V^\top (Z_{t-1} Z_{t-1}^\top)^{-1} U)^{-1} V^\top (Z_{t-1} Z_{t-1}^\top)^{-1}, \end{aligned}$$

for the time to compute the inverse, we note that

- Computing $(Z_{t-1} Z_{t-1}^\top)^{-1} U$ takes $O(k^2)$ time.
- Computing $V^\top (Z_{t-1} Z_{t-1}^\top)^{-1}$ takes $O(k^2)$ time.
- Computing $V^\top (Z_{t-1} Z_{t-1}^\top)^{-1} U$ takes $O(k^2)$ time.
- Computing $(I - V^\top (Z_{t-1} Z_{t-1}^\top)^{-1} U)^{-1}$ takes $O(1)$ time.

Hence, the matrix G_t takes $O(k^2)$ time.

By Claim A.3, $Z_t^\top \cdot G_t \cdot Z_t$ is a rank-4 update of $Z_{t-1}^\top \cdot G_{t-1} \cdot Z_{t-1}$, and can be computed in $O(k^2)$ time. Thus computing Q takes $O(k^2)$ time in total. □

5.6 Running Time of Sample

The purpose of this section is to prove Lemma 5.7, which shows the running time of the $\textsc{Sample}$ procedure.

Lemma 5.7 (Running time of Sample). *Procedure* $\textsc{Sample}$ *takes $Q \in \mathbb{R}^{|E| \times |E|}$ and $E \subseteq [d]$ as inputs and runs in $O(k^2 \log n + k^{\omega-2} d^2)$ time. Here ω is the exponent of matrix multiplication.*

Proof. Consider procedure $\textsc{Sample}(\mathcal{T}, Q, E)$. If $\mathcal{T}$ is a leaf, then by Lemma 5.8, its running time is $O(k^{\omega-2} d^2)$. If $\mathcal{T}$ is not a leaf, then the procedure first spends $O(k^2)$ time to compute p_ℓ and p_r, and then recursively calls one of $\textsc{Sample}(\mathcal{T}.\texttt{left}, Q, E)$ or $\textsc{Sample}(\mathcal{T}.\texttt{right}, Q, E)$. Since the depth of T is $O(\log(n/d)) = O(\log n)$, $\textsc{Sample}(T.\text{root}, Q, E)$ takes $O(\log n)$ layers of recursion until $\mathcal{T}$ is a leaf.

Thus, the running time of $\textsc{Sample}(Q, E)$ is: $O(k^2 \log n + k^{\omega-2} d^2)$. Thus, we complete the proof. □

Lemma 5.8. *Procedure* $\textsc{Sample\_Auxiliary}(\mathcal{T}, Q, E)$ *with $|\mathcal{T}.A| = d$ in Algorithm 2 takes $O(k^{\omega-2} d^2)$ time. Here ω is the exponent of matrix multiplication.*

Proof. $V_{E,:}$ is a $k \times d$ matrix, thus computing M takes

$$\begin{aligned} \mathcal{T}_{\mathrm{mat}}(d, k, k) + \mathcal{T}_{\mathrm{mat}}(d, k, d) &= O(d/k \cdot k^\omega + (d/k)^2 \cdot k^\omega) \\ &= O(k^{\omega-2} d^2) \end{aligned}$$

time.

Other operation all takes $O(k^2 d)$ time. Thus, the procedure takes $O(k^{\omega-2} d^2)$ time in total. □

6 DISCUSSION

In this paper, we study efficient algorithms to sample from the DPP and NDPP distributions when the input kernel matrix admits a low-rank decomposition. We show that it suffices to design data structures that support efficient sampling based on inner products. To achieve this goal, we either use a sampling tree that batches the input to speed up the preprocessing or use more novel and fast data structures to get superior query time. One limitation is that our approach, in its current form, may not be easily extended to improve the efficiency of DPP (NDPP) learning or inference. Thus, it is important to develop more efficient algorithms for these tasks.

Acknowledgements

We would like to thank Dangyang Zhuo, Aravind Reddy, Lianke Qin and Hang Hu for helpful discussions. Lichen Zhang was supported in part by NSF CCF-1955217 and NSF DMS-2022448.

References

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

Nima Anari, Shayan Oveis Gharan, and Alireza Rezaei. Monte carlo markov chain algorithms for sampling strongly rayleigh distributions and determinantal point processes. In *Conference on Learning Theory (COLT)*, 2016.

Erdem Biyik, Kenneth Wang, Nima Anari, and Dorsa Sadigh. Batch active learning using determinantal point processes. *arXiv:1906.07975*, 2019.

Victor-Emmanuel Brunel. Learning signed determinantal point processes through the principal minor assignment problem. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

Daniele Calandriello, Michał Dereziński, and Michal Valko. Sampling from a k-dpp without looking at all items. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

Elisa Celis, Vijay Keswani, Damian Straszak, Amit Deshpande, Tarun Kathuria, and Nisheeth Vishnoi. Fair and diverse dpp-based data summarization. In *International Conference on Machine Learning (ICML)*, 2018.

Michał Dereziński, Daniele Calandriello, and Michal Valko. Exact sampling of determinantal point processes with sublinear time preprocessing. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

Michał Dereziński, Rajiv Khanna, and Michael W Mahoney. Improved guarantees and a multiple-descent curve for column subset selection and the nystrom method. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020a.

Michał Dereziński, Feynman Liang, and Michael Mahoney. Bayesian experimental design using regularized determinantal point processes. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2020b.

Ran Duan, Hongxun Wu, and Renfei Zhou. Faster matrix multiplication via asymmetric hashing. In *FOCS*, 2023.

François Le Gall. Faster rectangular matrix multiplication by combination loss analysis. In *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 3765–3791. SIAM, 2024.

Mike Gartrell, Victor-Emmanuel Brunel, Elvis Dohmatob, and Syrine Krichene. Learning nonsymmetric determinantal point processes. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

Mike Gartrell, Insu Han, Elvis Dohmatob, Jennifer Gillenwater, and Victor-Emmanuel Brunel. Scalable learning and map inference for nonsymmetric determinantal point processes. In *International Conference on Learning Representations (ICLR)*, 2021.

Jennifer Gillenwater, Alex Kulesza, Zelda Mariet, and Sergei Vassilvtiskii. A tree-based method for fast repeated sampling of determinantal point processes. In *International Conference on Machine Learning (ICML)*, 2019.

Insu Han, Mike Gartrell, Jennifer Gillenwater, Elvis Dohmatob, and Amin Karbasi. Scalable sampling for nonsymmetric determinantal point processes. In *International Conference on Learning Representations (ICLR)*, 2022.

Tarun Kathuria and Amit Deshpande. On sampling and greedy map inference of constrained determinantal point processes. *arXiv preprint arXiv:1607.01551*, 2016.

Alex Kulesza and Ben Taskar. k-dpps: Fixed-size determinantal point processes. In *International Conference on Machine Learning (ICML)*, 2011.

Alex Kulesza and Ben Taskar. Determinantal point processes for machine learning. *Foundations and Trends® in Machine Learning*, 2012.

François Le Gall. Powers of tensors and fast matrix multiplication. In *Proceedings of the 39th international symposium on symbolic and algebraic computation (ISSAC)*, pages 296–303. ACM, 2014.

Chengtao Li, Stefanie Jegelka, and Suvrit Sra. Efficient sampling for k-determinantal point processes. In *Artificial Intelligence and Statistics (AISTATS)*, 2016a.

Chengtao Li, Stefanie Jegelka, and Suvrit Sra. Fast dpp sampling for nystrom with application to kernel methods. In *International Conference on Machine Learning (ICML)*, 2016b.

Chengtao Li, Stefanie Jegelka, and Suvrit Sra. Fast mixing markov chains for strongly rayleigh measures, dpps, and constrained sampling. In *Advances in Neural Information Processing Systems (NIPS)*, 2016c.

Zelda Mariet and Suvrit Sra. Diversity networks: Neural network compression using determinantal point processes. In *International Conference on Learning Representations (ICLR)*, 2016.

Takayuki Osogami, Rudy Raymond, Akshay Goel, Tomoyuki Shirai, and Takanori Maehara. Dynamic determinantal point processes. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2018.

Laura Perez-Beltrachini and Mirella Lapata. Multi-document summarization with determinantal point process attention. *Journal of Artificial Intelligence Research*, 71:371–399, 2021.

Jack Poulson. High-performance sampling of generic determinantal point processes. *Philosophical Transactions of the Royal Society A*, 378(2166):20190059, 2020.

Nicolas Tremblay, Simon Barthelmé, and Pierre-Olivier Amblard. Determinantal point processes for coresets. *J. Mach. Learn. Res.*, 20:168–1, 2019.

John Von Neumann. 13. various techniques used in connection with random digits. *Appl. Math Ser*, 12 (36-38):3, 1951.

Virginia Vassilevska Williams. Multiplying matrices faster than coppersmith-winograd. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing (STOC)*, pages 887–898. ACM, 2012.

Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. New bounds for matrix multiplication: from alpha to omega. In *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 3792–3835. SIAM, 2024.

Max A Woodbury. The stability of out-input matrices. *Chicago, IL*, 9, 1949.

Max A. Woodbury. *Inverting modified matrices*. Princeton University, Princeton, N. J., 1950. Statistical Research Group, Memo. Rep. no. 42.

Checklist

1. For all models and algorithms presented, check if you include:
   (a) A clear description of the mathematical setting, assumptions, algorithm, and/or model. [Yes]
   (b) An analysis of the properties and complexity (time, space, sample size) of any algorithm. [Yes]
   (c) (Optional) Anonymized source code, with specification of all dependencies, including external libraries. [Not Applicable]

2. For any theoretical claim, check if you include:
   (a) Statements of the full set of assumptions of all theoretical results. [Yes]
   (b) Complete proofs of all theoretical results. [Yes]
   (c) Clear explanations of any assumptions. [Yes]

3. For all figures and tables that present empirical results, check if you include:
   (a) The code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL). [Not Applicable]
   (b) All the training details (e.g., data splits, hyperparameters, how they were chosen). [Not Applicable]
   (c) A clear definition of the specific measure or statistics and error bars (e.g., with respect to the random seed after running experiments multiple times). [Not Applicable]
   (d) A description of the computing infrastructure used. (e.g., type of GPUs, internal cluster, or cloud provider). [Not Applicable]

4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets, check if you include:

   (a) Citations of the creator If your work uses existing assets. [Not Applicable]
   (b) The license information of the assets, if applicable. [Not Applicable]
   (c) New assets either in the supplemental material or as a URL, if applicable. [Not Applicable]
   (d) Information about consent from data providers/curators. [Not Applicable]
   (e) Discussion of sensible content if applicable, e.g., personally identifiable information or offensive content. [Not Applicable]

5. If you used crowdsourcing or conducted research with human subjects, check if you include:
   (a) The full text of instructions given to participants and screenshots. [Not Applicable]
   (b) Descriptions of potential participant risks, with links to Institutional Review Board (IRB) approvals if applicable. [Not Applicable]
   (c) The estimated hourly wage paid to participants and the total amount spent on participant compensation. [Not Applicable]

Appendix

Roadmap. In Section A, we present several basic tools, which include the fundamental claims and the definition of rank-k update. In Section B, we present our fast NDPP sampling data structure. In Section C, we provide an additional discussion about data structures and analyze their key properties.

A BASIC TOOLS

In this section, we introduce some basic tools.

Definition A.1 (Rank-k Update)**.** *Let A and B be $d \times d$ matrices. We say B is a rank-k update of A if B can be expressed as*

$$A + u_1 v_1^\top + u_2 v_2^\top + \cdots + u_k v_k^\top,$$

where u_i, v_i are both d-dimensional vector, for every $i \in [k]$.

Claim A.2. *Let A be a $d \times d$ matrix. If A^{new} is a rank-k update of A, then $(A^{\mathrm{new}})^\top$ is a rank-k update of $A^\top$.*

Proof. Suppose $A^{\mathrm{new}} = A + u_1 v_1^\top + \cdots + u_k v_k^\top$, then $(A^{\mathrm{new}})^\top = A^\top + v_1 u_1^\top + \cdots + v_k u_k^\top$ is a rank-k update of $A^\top$. □

Claim A.3. *Let A, B be $d \times d$ matrices. If A^{new} is rank-k update of A and B^{new} is rank-l update of B, then $A^{\mathrm{new}} B^{\mathrm{new}}$ is rank-$(k+l)$ update of AB.*

Proof. Suppose

$$A^{\mathrm{new}} = A + \Delta_A,$$

in which Δ_A is rank k and

$$B^{\mathrm{new}} = B + \Delta_B,$$

in which Δ_B is rank l, then

$$A^{\mathrm{new}} B^{\mathrm{new}} = AB + \Delta_A B + A \Delta_B + \Delta_A \Delta_B.$$

Using the fact that $\mathrm{rank}(XY) \le \min\{\mathrm{rank}(X), \mathrm{rank}(Y)\}$, we get

$$\begin{aligned} \mathrm{rank}(\Delta_A B) \le & \ \min\{\mathrm{rank}(\Delta_A), \mathrm{rank}(B)\} = k \\ \mathrm{rank}((A + \Delta_A)\Delta_B) \le & \ \min\{\mathrm{rank}(A), \mathrm{rank}(\Delta_B)\} = l \end{aligned}$$

Thus, $A^{\mathrm{new}} B^{\mathrm{new}}$ is a rank-$(k+l)$ update of AB. □

Claim A.4. *Let A be a $d \times d$ invertible matrix. If A^{new} is a rank-k update of A and A^{new} is also invertible, then $(A^{\mathrm{new}})^{-1}$ is a rank-k update of A^{-1}.*

Proof. We prove this by using the matrix Woodbury formula.

Suppose $A^{\mathrm{new}} = A + UV^\top$ for which $U, V \in \mathbb{R}^{d \times k}$, then

$$\begin{aligned} (A^{\mathrm{new}})^{-1} &= (A + UV^\top)^{-1} \\ &= A^{-1} - A^{-1} U (I - V^\top A^{-1} U)^{-1} V^\top A^{-1} \end{aligned}$$

note the smallest rank among the product is k, hence the update is rank k. □

B FAST NDPP SAMPLING DATA STRUCTURE

In this section, we show that our fast DPP sampling algorithm implies a fast NDPP sampling algorithm.

Corollary B.1 (NDPP Sampling Data Structure)**.** *Let $k \leq d$ be a positive integer. There exists a data structure that uses $O(nd)$ space with the following procedures:*

- Init$(\{\lambda_i, u_i\}_{i=1}^d, n \in \mathbb{N}_+, d \in \mathbb{N}_+)$. *Taken the eigendecomposition $\{\lambda_i, u_i\}_{i=1}^d$ of kernel $\hat{L}$ as input where $\lambda_i \in \mathbb{R}, u_i \in \mathbb{R}^n$, the data structure initializes in $O(nd^{\omega-1})$ time.*
- Query(). *It outputs a list of indices $Y \subset [n]$ where $|Y| = k$ following NDPP distribution in time $O((k^3 \log n + k^{\omega-1} d^2)(1+w)^{d/2})$, where w is a data-dependent parameter*[3].

Proof. The algorithm pseudocode is presented in Algorithm 4. The main idea follows from rejection sampling the NDPP distribution of L from the DPP distribution of $\hat{L}$.

As shown in Theorem 1 in Han et al. (2022), the probability ratio for a subset $Y \subset [n]$ is:

$$\frac{\Pr_L[Y]}{\Pr_{\hat{L}}[Y]} = \frac{\det(L_Y)/\det(\hat{L}+I)}{\det(\hat{L}_Y)/\det(\hat{L}+I)} \leq \frac{\det(\hat{L}+I)}{\det(L+I)}.$$

Hence, there exists a universal constant $U \leq \frac{\det(\hat{L}+I)}{\det(L+I)}$ that upper bounds the ratio between the target distribution $\Pr_L[Y]$ and the proposed distribution $\Pr_{\hat{L}}[Y]$. Therefore, the rejection sampling gives that, if we take the acceptance probability to be $p := \frac{\det(L_Y)}{\det(\hat{L}_Y)}$, then the resulting distribution will be the target distribution: the NDPP with matrix L. Thus, the sample correctness of our algorithm is proved.

For the running time of Algorithm 4, we note that the NDPP matrix L is implicitly given as input to the algorithm, since we only need query access to it. For the Init procedure, its time complexity is dominated by the time complexity of the DPP sampling data structure. Thus, by Lemma 5.4, we know that the running time is $O(nd^{\omega-1})$.

For the Query procedure, Han et al. (2022) showed that the number of iterations of the rejection sampling is bounded by $(1+w)^{d/2}$, where $w = \frac{2}{d}\sum_{i=1}^{d/2} \frac{2\sigma_i}{\sigma_i^2+1} \in (0,1]$ and σ_i are the singular values of the skew-symmetric part of L. In each iteration, the DPP sample Y can be obtained in $O(k^3 \log n + k^{\omega-1} d^2)$-time by Lemma 5.5. The matrix $\hat{L}_Y \in \mathbb{R}^{k \times k}$ can be formed in $O(k^2 d)$-time and the quantity p can be computed in $O(k^\omega)$-time. Therefore, the total running time is

$$(1+w)^{d/2} \cdot O(k^3 \log n + k^{\omega-1} d^2 + k^2 d + k^\omega) = O((k^3 \log n + k^{\omega-1} d^2)(1+w)^{d/2}),$$

where we use the condition of $k \leq d$.

The corollary is then proved. ☐

C ADDITIONAL DATA STRUCTURES: A DISCUSSION

In this section, we provide two more data structures that can see potential usage for DPP and NDPP sampling. In Section C.1, the first data structure is related to the classic prefix sum array (see Algorithm 5 and 6). In Section C.2, the second data structure is a sophisticated data structure that uses coefficients to maintain a structured projection matrix (see Algorithm 7).

C.1 Compute Prefix Inner Products: Sum Array

In this section, we present a Sum Array data structure. Roughly speaking, we first compute all the inner products via a matrix multiplication, then accumulate a sum array containing all the prefix sums of inner products. This way, we can query any range of inner products in $O(1)$ time.

[3]As observed in Han et al. (2022), in many practical datasets, the factor $(1+w)^d \ll n$.

Algorithm 4 Fast NDPP sampling algorithm

```
1: data structure NDPPSampling
2: members
3:     Data structure EfficientPrepDPP DPP                                  ▷ Algorithm 1
4:     Query access to the NDPP matrix $L$
5:     NDPP kernel $\hat{L}$'s eigendecomposition $\{\lambda_i, u_i\}_{i=1}^d$
6: end members
7: procedure Init($L$, $\{\lambda_i, u_i\}_{i=1}^d$)
8:     $L \leftarrow L$, $\{\lambda_i, u_i\}_{i=1}^d \leftarrow \{\lambda_i, u_i\}_{i=1}^d$      ▷ Setup query access to $L$
9:     DPP.Init($\{\lambda_i, u_i\}_{i=1}^d$)                               ▷ It takes $O(nd^{\omega-1})$-time
10: end procedure
11:
12: procedure Query
13:     while true do
14:         $Y \leftarrow$ DPP.Query()                          ▷ It takes $O(k^3 \log n + k^{\omega-1} d^2)$-time
15:         $\hat{L}_Y \leftarrow \sum_{i=1}^d \lambda_i (u_i)_Y (u_i)_Y^\top$                     ▷ It takes $O(k^2 d)$-time
16:         $p \leftarrow \frac{\det(L_Y)}{\det(\hat{L}_Y)}$                                  ▷ It takes $O(k^\omega)$-time
17:         $u \sim \mathrm{Uniform}([0,1])$
18:         if $u \leq p$ then
19:             return $Y$
20:         end if
21:     end while
22: end procedure
```

Algorithm 5 Sum Arrary in one-dimensional

```
1: data structure SumArraryOneD
2: members
3:     A vector $x \in \mathbb{R}^{n+1}$
4:     function $f : \mathbb{R} \to \mathbb{R}$
5: end members
6:
7: procedure Init($u \in \mathbb{R}^d, \{v_1, v_2, \cdots, v_n\} \subset \mathbb{R}^d, f : \mathbb{R} \to \mathbb{R}$)
8:                                                                 ▷ This takes $O(nd)$ time
9:     $f \leftarrow f$
10:    $x \leftarrow \{0\}^{n+1}$                  ▷ For convenient, we use the index of $x$ from 0 to $n$
11:    $x_0 \leftarrow 0$
12:    for $i = 1$ to $n$ do
13:        $x_i \leftarrow x_{i-1} + f(\langle u, v_i \rangle)$
14:    end for
15: end procedure
16:
17: procedure Query($a \in [n], b \in [n]$)
18:                                                                 ▷ We require that $a < b$
19:     return $x_b - x_{a-1}$                            ▷ For simplicity, we assume $x_0 \leftarrow 0$
20: end procedure
21: end data structure
```

Theorem C.1. *There exists a data structure* SumArraryOneD *(Algorithm 5) uses $O(n + d)$ space with the following procedures:*

- Init$(u \in \mathbb{R}^d, \{v_1, v_2, \cdots, v_n\} \subseteq \mathbb{R}^d, f : \mathbb{R} \to \mathbb{R})$. *Procedure* Init *takes a vector u, a set of vectors $\{v_1, v_2, \cdots, v_n\}$ and a function $f : \mathbb{R} \to \mathbb{R}$ as input and initializes in $O(nd)$ time.*
- Query$(a \in [n], b \in [n])$. *Procedure* Query *takes two indices $a, b \in [n]$ as input, returns $\sum_{i=a}^b f(u, v_i)$ in*

$O(1)$ time.

Proof. For procedure INIT, each for-loop takes $O(d)$ time to compute inner product, thus INIT takes $O(nd)$ time in total.

For procedure QUERY(a, b), for correctness, we have it outputs

$$\sum_{i=1}^{b} f(\langle u, v_i \rangle) - \sum_{i=1}^{a} f(\langle u, v_i \rangle) = \sum_{i=a}^{b} f(\langle u, v_i \rangle),$$

for running time, since getting x_b and x_a takes $O(1)$ time, we have it runs in $O(1)$ time. $\square$

Theorem C.2. *There exists a data structure* SUMARRAYTWOD *(Algorithm 6) uses $O(n^2)$ space with the following procedures:*

- INIT$(\{v_1, v_2, \cdots, v_n\} \subseteq \mathbb{R}^d, f : \mathbb{R} \to \mathbb{R})$. *Procedure* INIT *takes a set of vectors $\{v_1, v_2, \cdots, v_n\}$ and a function $f : \mathbb{R} \to \mathbb{R}$ as input and initializes in $\mathcal{T}_{\mathrm{mat}}(n, d, n)$ time.*
- QUERY$(i \in [n], a \in [n], b \in [n])$. *Procedure* QUERY *takes three indices $i \in [n]$, $a \in [n]$ and $b \in [n]$ as input, returns $\sum_{j=a}^{b} f(\langle v_i, v_j \rangle)$ in $O(1)$ time.*

Proof. For procedure INIT, we need to

- Compute $Y = VV^\top$, which takes $\mathcal{T}_{\mathrm{mat}}(n, d, n)$ time.
- Construct the matrix X, since it's an $(n + 1) \times (n + 1)$ time, and construct each entry requires to evaluate f, suppose the evaluation of f is $O(1)$, this takes $O(1)$ time.

Hence, the time is dominated by $\mathcal{T}_{\mathrm{mat}}(n, d, n)$.

For procedure QUERY, we just need to read two entries of X, which takes $O(1)$ time. $\square$

Remark C.3. *The two dimensional sum array data structure can effectively handle the problem that makes use of the form of $f(\langle v_i, v_j \rangle)$. In the case of DPP sampling, we need to construct a matrix Z iteratively that consists of rows of a known matrix V, and at each iteration, compute the inner product between rows of V and a projection matrix defined by Z: $Z^\top (ZZ^\top)^{-1} Z$. Note that the information we need here is essentially the inner product square $\langle v_i, v_j \rangle^2$. Hence, the two dimensional sum array data structure can see potential usage in improving DPP sampling process.*

C.2 The FindCoefficients Procedure

In this section, we study another data structure that handles a specific type of projection maintenance and quadratic form computations.

Suppose we are given a matrix $X \in \mathbb{R}^{n \times d}$ in advance, and we are iteratively constructing a matrix $Z \in \mathbb{R}^{k \times d}$ that consists of a subset of X's rows with cardinality k. At each iteration, we need to compute the following quadratic form: let $x_i \in \mathbb{R}^d$ be the i-th row of X, then we are asked to compute

$$x_i^\top \underbrace{Z^\top (ZZ^\top)^{-1} Z}_{P} x_i.$$

Note that even the projection matrix $P \in \mathbb{R}^{d \times d}$ is given, naively forming the quadratic form still takes $O(d^2)$ time.

We present a data structure that takes $\mathcal{T}_{\mathrm{mat}}(n, d, n)$ time for preprocessing, and then support the one-quadratic form query in time $O(kd)$.

We start by introducing some definitions.

Algorithm 6 Sum arrary in two-dimensional

```
1: data structure SumArraryTwoD                                      ▷ Theorem C.2
2: members
3:     A matrix $X \in \mathbb{R}^{(n+1)\times(n+1)}$
4:     function $f : \mathbb{R} \to \mathbb{R}$
5: end members
6:
7: procedure Init($\{v_1, v_2, \cdots, v_n\} \subset \mathbb{R}^d, f : \mathbb{R} \to \mathbb{R}$)
8:                                                ▷ This takes $\mathcal{T}_{\mathrm{mat}}(n, d, n)$ time
9:     $V \leftarrow [\ v_1\ v_2\ \cdots\ v_n\ ]^\top$                              ▷ $V \in \mathbb{R}^{n\times d}$
10:    Compute $Y = V \cdot V^\top$
11:    $f \leftarrow f$
12:    $X \leftarrow \{0\}^{(n+1)\times(n+1)}$      ▷ For convenient, we use the index of $x$ from 0 to $n$
13:    for $i = 1 \to n$ do
14:        $X_{i,0} \leftarrow 0$
15:        for $j = 1 \to n$ do
16:            $X_{i,j} \leftarrow X_{i,j-1} + f(Y_{i,j})$
17:        end for
18:    end for
19: end procedure
20:
21: procedure Query($i \in [n], a \in [n], b \in [n]$)
22:                                                        ▷ We require that $a < b$
23:    return $X_{i,b} - X_{i,a-1}$                ▷ For simplicity, we assume $X_{i,0} \leftarrow 0$
24: end procedure
25: end data structure
```

Definition C.4. *Given d-dimensional vectors $x_1, x_2, \cdots, x_n$, for $t \in [n]$, we define $Z_t = [\ x_1\ x_2\ \cdots\ x_t\ ]^\top \in \mathbb{R}^{t\times d}$, and $H_t = Z_t^\top (Z_t Z_t^\top)^{-1} Z_t \in \mathbb{R}^{d\times d}$.*

We first show that the $d \times d$ matrix H_t can be expressed in a special form.

Lemma C.5. *For $k \in [n]$,*

$$H_k = Z_k^\top (Z_k Z_k^\top)^{-1} Z_k = \underbrace{\sum_{i=1}^{k} \alpha_i x_i x_i^\top}_{\text{diagonal terms}} + \underbrace{\sum_{i\neq j} \alpha_{i,j} x_i x_j^\top}_{\text{off}-\text{diagonal terms}} ,$$

where $H_k \in \mathbb{R}^{d\times d}$ and $Z_k \in \mathbb{R}^{t\times d}$ are matrices defined in Definition C.4.

Proof. We will prove by induction on k.

Base case. When $k = 1$, then the target expression is

$$x_1 (x_1^\top x_1)^{-1} x_1^\top = \frac{1}{\|x_1\|_2^2} x_1 x_1^\top .$$

Inductive hypothesis. Suppose up to some $k < d$, we have that

$$(Z_k)^\top (Z_k (Z_k)^\top)^{-1} Z_k = \sum_{i=1}^{k} \alpha_i x_i x_i^\top + \sum_{i\neq j} \alpha_{i,j} x_i x_j^\top$$

Inductive step. Now, let's prove for $k+1$. First, observe that

$$\begin{aligned} Z_{k+1}Z_{k+1}^\top = & \begin{bmatrix} x_1^\top x_1 & x_1^\top x_2 & \dots & x_1^\top x_k & x_1^\top x_{k+1} \\ \vdots & \vdots & \dots & \vdots & \vdots \\ x_k^\top x_1 & x_k^\top x_2 & \dots & x_k^\top x_k & x_k^\top x_{k+1} \\ x_{k+1}^\top x_1 & x_{k+1}^\top x_2 & \dots & x_{k+1}^\top x_k & x_{k+1}^\top x_{k+1} \end{bmatrix} \\ = & \begin{bmatrix} A & b \\ b^\top & c \end{bmatrix} \end{aligned}$$

where $A = Z_k Z_k^\top \in \mathbb{R}^{k\times k}$, $b = \begin{bmatrix} x_1^\top x_{k+1} \\ \vdots \\ x_k^\top x_{k+1} \end{bmatrix} \in \mathbb{R}^k$ and $c = x_{k+1}^\top x_{k+1}$. We also use $d = (c - b^\top A^{-1} b)^{-1} \in \mathbb{R}$.

Using Schur complement, we have

$$\begin{aligned} (Z_{k+1}Z_{k+1}^\top)^{-1} = & \left(\begin{bmatrix} A & b \\ b^\top & c \end{bmatrix} \right)^{-1} \\ = & \begin{bmatrix} A^{-1} + A^{-1}b(c-b^\top A^{-1}b)^{-1}b^\top A^{-1} & -A^{-1}b(c-b^\top A^{-1}b)^{-1} \\ -(c-b^\top A^{-1}b)^{-1}b^\top A^{-1} & (c-b^\top A^{-1}b)^{-1} \end{bmatrix} \\ = & \begin{bmatrix} A^{-1} + dA^{-1}bb^\top A^{-1} & -dA^{-1}b \\ -db^\top A^{-1} & d \end{bmatrix} \end{aligned}$$

$$\begin{aligned} Z_{k+1}^\top (Z_{k+1}Z_{k+1}^\top)^{-1} Z_{k+1} = & \begin{bmatrix} Z_k^\top & x_{k+1} \end{bmatrix} \begin{bmatrix} A^{-1} + dA^{-1}bb^\top A^{-1} & -dA^{-1}b \\ -db^\top A^{-1} & d \end{bmatrix} \begin{bmatrix} Z_k \\ x_{k+1}^\top \end{bmatrix} \\ = & \begin{bmatrix} Z_k^\top & x_{k+1} \end{bmatrix} \begin{bmatrix} A^{-1}Z_k + dA^{-1}bb^\top A^{-1}Z_k - dA^{-1}bx_{k+1}^\top \\ -db^\top A^{-1}Z_k + dx_{k+1}^\top \end{bmatrix} \\ = & Z_k^\top A^{-1} Z_k + dZ_k^\top A^{-1}bb^\top A^{-1}Z_k - dZ_k^\top A^{-1}bx_{k+1}^\top \\ & - dx_{k+1}b^\top A^{-1}Z_k + dx_{k+1}x_{k+1}^\top. \end{aligned}$$

We compute each term individually.

- For $Z_k^\top A^{-1} Z_k \in \mathbb{R}^{d\times d}$, by inductive hypothesis, we know it is $\sum_{i=1}^k \alpha_i x_i x_i^\top + \sum_{i\neq j} \alpha_{i,j} x_i x_j^\top$.
- For $dZ_k^\top A^{-1}bb^\top A^{-1}Z_k \in \mathbb{R}^{d\times d}$. For $j \in [k]$, let a_j denote the j-th row of the $k\times k$ matrix A^{-1}, then we know that $Z_k^\top A^{-1} = \sum_{j=1}^k x_j a_j^\top \in \mathbb{R}^{d\times d}$, and

$$\begin{aligned} Z_k^\top A^{-1} b = & \sum_{j=1}^k x_j a_j^\top b \\ = & \sum_{j=1}^k (a_j^\top b) x_j \end{aligned}$$

  is a d-dimensional vector. The outer product gives

$$\begin{aligned} Z_k^\top A^{-1} b (Z_k^\top A^{-1} b)^\top = & (\sum_{j=1}^k (a_j^\top b) x_j) \cdot (\sum_{j=1}^k (a_j^\top b) x_j^\top) \\ = & \sum_{i=1}^k (a_i^\top b)^2 x_i x_i^\top + \sum_{i\neq j} (a_i^\top b)(a_j^\top b) x_i x_j^\top \end{aligned}$$

- For $dZ_k^\top A^{-1} b x_{k+1}^\top \in \mathbb{R}^{d\times d}$, recall that $Z_k^\top A^{-1} b = \sum_{j=1}^k (a_j^\top b) x_j$ is a d-dimensional vector, therefore

$$\begin{aligned} Z_k^\top A^{-1} b x_{k+1}^\top &= (\sum_{j=1}^k (a_j^\top b) x_j) x_{k+1}^\top \\ &= \sum_{i=1}^k (a_i^\top b) x_i x_{k+1}^\top \end{aligned}$$

- For $dx_{k+1} b^\top A^{-1} Z_k \in \mathbb{R}^{d\times d}$, it is symmetric as the prior case.
- For $dx_{k+1} x_{k+1}^\top \in \mathbb{R}^{d\times d}$, it is straightforward.

Hence, we have shown that we can express H_k as a sum of the outer product of x_i's, as desired. □

Definition C.6. *For a matrix $M \in \mathbb{R}^{d\times d}$, if M can be expressed as $\sum_{i=1}^k \alpha_i x_i x_i^\top + \sum_{i\neq j} \alpha_{i,j} x_i x_j^\top$, we call the set $\{\alpha_1, \cdots \alpha_k\} \cup \{\alpha_{i,j}\}_{i\neq j, i,j\in[k]}$ as the coefficient set of M.*

Remark C.7. *Note the above proof also provides an algorithm (see Algorithm 7) to find coefficients, suppose a new vector is added in and we have access to the inverse matrix of the last iteration.*

Theorem C.8 (Find Coeffeicients)**.** *There is an algorithm (see Algorithm 7) which takes the coefficient set of $H_k \in \mathbb{R}^{d\times d}$ and $(Z_k Z_k^\top)^{-1} \in \mathbb{R}^{k\times k}$ as input, and returns the coefficient set of $H_{k+1} \in \mathbb{R}^{d\times d}$ and $(Z_{k+1} Z_{k+1}^\top)^{-1}$ in $O(k^2 + kd)$ time.*

Lemma C.9 (Running Time)**.** *Algorithm 7 runs in $O(k^2 + kd)$ time.*

Proof. Computing c takes $O(d)$ time. Computing Z_k takes $O(kd)$ time. Computing b takes $O(kd)$ time. Computing g takes $O(kd)$ time. From line 8 to line15, the algorithm computes $\alpha_{i,j}$ for every $i, j \in [k+1]$, taking $O(1)$ time respectively, that is, taking $O(k^2)$ time in total. Computing A_{k+1}^{-1} takes $O(k^2)$ time.

To sum up, Algorithm 7 runs in $O(k^2 + kd)$ time. □

Algorithm 7 The FindCoefficients procedure, it takes the representation from last iteration and computes a new representation.

```
1: procedure FindCoefficients($\{\alpha_{i,j}\}_{(i,j)\in[k]\times[k]}$, $\{x_1, \ldots, x_k, x_{k+1}\} \subset \mathbb{R}^d$, $A_k^{-1} \in \mathbb{R}^{k\times k}$)
2:                                   ▷ $\{\alpha_{i,j}\}$ are coefficient sets of $H_k$, $(Z_k Z_k^\top)^{-1}$ respectively
3:                                   ▷ $A^{-1}$ is $(Z_k Z_k^\top)^{-1}$
4:     $c \leftarrow x_{k+1}^\top x_{k+1}$
5:     $Z_k \leftarrow \begin{bmatrix} x_1 & x_2 \cdots x_k \end{bmatrix}^\top \in \mathbb{R}^{k\times d}$
6:     $b \leftarrow Z_k x_{k+1} \in \mathbb{R}^k$
7:     $g \leftarrow A_k^{-1} b \in \mathbb{R}^k$
8:     $\alpha_{k+1,k+1} \leftarrow (c - b^\top A_k^{-1} b)^{-1}$
9:     for $i = 1 \to k$ do
10:        $\alpha_{i,k+1} \leftarrow \alpha_{k+1} \cdot g_i$
11:        $\alpha_{k+1,i} \leftarrow \alpha_{k+1} \cdot g_i$
12:        for $j = 1 \to k$ do
13:            $\alpha_{i,j} \leftarrow \alpha_{i,j} + \alpha_{k+1} \cdot g_i \cdot g_j$
14:        end for
15:    end for
16:    $A_{k+1}^{-1} = \begin{bmatrix} A_k^{-1} + dA_k^{-1} b b^\top A_k^{-1} & -dA_k^{-1} b \\ -db^\top A_k^{-1} & d \end{bmatrix}$
17:    return $\{\alpha_{i,j}\}_{i,j\in[k+1]}, A_{k+1}^{-1}$
18: end procedure
```

Remark C.10. *We now give an overview of how to implement a data structure with* FindCoefficients*. In preprocessing, we simply compute the matrix multiplication $XX^\top$ in time $\mathcal{T}_{\mathrm{mat}}(n, d, n)$, notice this product stores all pairwise inner products between rows of X.*

Given a row $x_i \in \mathbb{R}^d$ and a matrix $Z \in \mathbb{R}^{k \times d}$ that contains k rows of X, we observe that

$$\begin{aligned} & x_i^\top (\sum_{j=1}^k \alpha_i x_j x_j^\top + \sum_{j \neq l} \alpha_{i,j} x_j x_l^\top) x_i \\ = & \sum_{j=1}^k \alpha_i \langle x_i, x_j \rangle^2 + \sum_{j \neq l, (j,l) \in [k] \times [k]} \alpha_{i,j} \langle x_i, x_j \rangle \cdot \langle x_i, x_l \rangle \end{aligned}$$

Since the inner products have been pre-computed, we only need to find the coefficient set $\{\alpha_1, \ldots, \alpha_k\} \cup \{\alpha_{j,l}\}_{j \neq l, j, l \in [k]}$.

The FindCoefficients *procedure tells us that if Z is iteratively constructed by inserting one new row at a time, then we can find the coefficients in time $O(kd)$. Updating all $O(k^2)$ coefficients takes time $O(k^2)$. Hence, the overall time of computing this quadratic form is $O(kd + k^2)$.*

Remark C.11. *We notice that the DPP sampling procedure is essentially building up the matrix Z and computing the quadratic form at each iteration. If one can generalize this data structure to handle computing n quadratic forms efficiently, then it will lead to direct improvement in the DPP process.*