Counting linear extensions with volume computation

By

Reyan Ahmed

Mentors:

Vissarion Fisikopoulos
Elias Tsigaridas
Matias Bender

Final submission report

Google Summer of Code
GSoC
&
Volume Estimation
GeomScale

September 9, 2022

1 Introduction

We have worked on the problem of counting linear extensions. In this problem of counting the linear extensions, we are given an initial partial order P, which is also known as partially ordered set (poset). The objective of this problem is to generate all orders (linear extensions) that preserve P. However, there can be an exponential number of such orders. Hence the running time to count the number of total orders that preserves P will be exponential. To get rid of exponential computation, we will approximate the number of linear extensions using volume computation. We will develop different approximation algorithms and compare their performance.

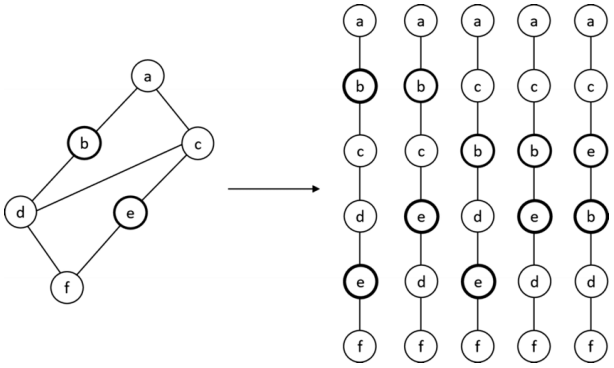


Figure 1: An example of linear extensions. On the left side, we see a partially ordered set (poset). There is no relation between the elements b and c. On the right side, we see all six complete/total orders of the poset.

2 Related Work

Counting linear extensions has many important applications [8, 9]. The fastest known dynamic programming requires exponential time and space [3, 5]. A dynamic programming relies on a recursive formulation of the problem. Several methods have been proposed for counting linear extensions by recursively decomposing the task into subproblems. Each linear extension of P begins with some minimal element $x \in P$, and the number of extensions that begin with x equals $\ell(P \backslash x)$. Therefore, we have that,

$$\ell(P) = \sum_{x \in \min(P)} \ell(P \backslash x) \tag{1}$$

where $\min(P)$ denotes the set of minimal elements of P.

For an arbitrary element $x \in P$, we say that a partition of $P \backslash x$ into a pair of sets (D, U) is admissible if D is a downset that contains all predecessors of

x (and U an upset that contains all successors of x). Equivalently, a partition is admissible if and only if there is at least one linear extension σ such that $\sigma(d) < \sigma(x) < \sigma(u)$ for all $d \in D$ and $u \in U$. Choosing a linear extension of P is equivalent to choosing such a partition and ordering D and U independently. Thus, we have that

$$\ell(P) = \sum_{(D,U)} \ell(D) \cdot \ell(U) \tag{2}$$

where (D, U) runs over all admissible partitions.

We say that a non-empty set of elements $S \subset P$ is a static set if every element in S is comparable with every element in $P \backslash S$ and if no proper subset of S has this property. It is known [7] that either P has no static sets, or there exists a unique partition of P into static sets $S_1, \cdots, S_k$. If the partition exists, a linear extension is obtained by ordering each S_i independently, and therefore it holds that

$$\ell(P) = \prod_{i=1}^{k} \ell(S_i) \tag{3}$$

If the graph representation of P is disconnected, i.e., if P can be partitioned into sets A and B such that a and b are incomparable for all $a \in A$ and $b \in B$, then taking a linear of extension of P is equivalent to ordering A and B independently and then interleaving them. Thus, in this case we have

$$\ell(P) = \ell(A) \cdot \ell(B) \cdot \binom{|P|}{|A|} \tag{4}$$

There are several MCMC based approximations that have been proposed [1, 11]. The latter improvements were based on rapidly mixing Markov chains in the set of linear extensions [6]. In practice, approximate counting of linear extensions can take hours for a few hundred elements [11]. It is well known that we can represent a partial order with a convex polyhedron, namely order polytope [10]. Furthermore, the number of linear extensions is equal to the volume of the order polytope. To the best of our knowledge, there is no algorithm that counts linear extensions through volume calculations of order polytopes. However, there are a few practical algorithms for estimating the volume of a convex polytope; Volesti provides the most efficient implementations [4].

3 Preliminaries

A *partially ordered set* (also called a *poset*) is a set P equipped with a binary relation $\leq$ satisfying the following three properties:

1. If $x \in P$, then $x \leq x$ in P (*reflexive* property).
2. If $x, y \in P, x \leq y$ in P and $y \leq x$ in P then $x = y$ (*antisymmetric* property).

3. If $x, y, z \in P, x \leq y$ in P and $y \leq z$ in P, then $x \leq z$ in P (*transitive* property).

Let X be a set. A *binary relation* P is a subset of $X \times X$. If P is reflexive then $\forall x \in X, (x, x) \in P$.

We can define different relational operations similar to the $\leq$ operation. For example, $x < y$ in P means $x \leq y$ and $x \neq y$. Also, $y > x$ in P means the same as $x < y$. Similarly, $x \leq y$ means the same as $y \geq x$. One can define these operators in a way different than the traditional way. For example, when P is a collection of sets, set $x \leq y$ in P when x is a subset of y. In this poset $\{2, 5\} < \{2, 5, 7, 8\}$ and $\{5, 8, 9\} \geq \{5, 8, 9\}$. When P is a set of positive integers, set $\leq$ in P when x divides y without remainder. In this poset, $15 < 105$ and $12 < 48$. However, 17 is not less than $1,000,000,000$.

The familiar binary operation $\leq$ on number systems like $\mathbb{Z}$ (integers), $\mathbb{Q}$ (rationals), $\mathbb{R}$ (reals) is a partial order. However, in each of these three cases, the binary relation $\leq$ satisfies a fourth condition: for all x, y either $x \leq y$ in P or $y \leq x$ in P. Partial order satisfying this condition are called *linear* orders or *total* orders.

When x and y are distinct points in a poset P, we say that x is *covered* by y in P when $x < y$ in P and there is no point z with $x < z < y$ in P. Alternatively, we may say that y *covers* x in P. With inclusion, $\{2, 5\}$ is covered by $\{2, 5, 7\}$ but $\{4, 6, 7\}$ is not covered by $\{4, 6, 7, 9, 11, 12\}$. With division, 15 is covered by 105 because $105 = 15 \times 7$. However, 14 is not covered by by 84 since $84 = 14 \times 2 \times 3$ (14×2 and 14×3 are between 14 and 84).

When P is a poset, we associate with P a graph G called the *cover graph* of P. The vertices of G are the vertices of P. We connect two vertices u, v in G if either u covers v or v covers u. If G is the cover graph of a poset P, a straight line drawing of G is called an *ordered diagram* if u is higher in the plane than v whenever u covers v in P.

Given a poset P, two distinct elements x, y are *comparable* when either $x < y$ or $y < x$ in P. In the *comparability graph* of P the vertex set is equal to the elements of P. Two distinct elements x, y are adjacent in the comparable graph if x and y are comparable in P. Similarly we can define the *incomparability graph*.

A *chain* is a subset in which every element is comparable. A chain is *maximal* if no superset is a chain. The *height* of a poset is the size of a chain that has the maximum number of elements. An *antichain* is a subset in which every element is incomparable. An antichain is *maximal* if no superset is an antichain. The *width* of a poset is the size of an antichain that has the maximum number of elements.

4 Our approach

We have used volume estimation to approximate the number of linear extensions of a poset. We first briefly talk about the idea of volume estimation. A classical

example of this technique is the approximation of π. Let us assume we have a random number generator and we want to estimate the value of π. Each time we uniformly sample to numbers from the range $[-1, 1]$. The pair of numbers can be viewed as a coordinate inside the square $[-1, 1] \times [-1, 1]$. We can draw a circle inside the square such that the perimeter of the circle touches the boundary of the circle. The radius of the circle is equal to one. Let us sample a pair of points inside the square. We can consider the point as a vector from the origin. If the length of the vector is less than or equal to one, then the point is inside the circle. Otherwise it is outside the circle.

We now draw random points many times while counting the number of times the point falls inside the circle. Since we are generating the points uniformly from the square, the ratio of the number of times the circle falls inside the circle to the total number of samples should be approximately equal to the ratio of the area of the circle to the area of the square. The more samples we take, the more accurate the approximation will be. Now the ratio of the circle is πr^2, where r is the radius of the circle. Recall that $r = 1$, hence the area of the circle is π. On the other hand, the area of the square is 4. Hence, we have the following observation:

$$\frac{\pi}{4} = \frac{\text{Number of samples inside circle}}{\text{Total number of samples}}$$

$$\pi = \frac{4 \times \text{Number of samples inside circle}}{\text{Total number of samples}}$$

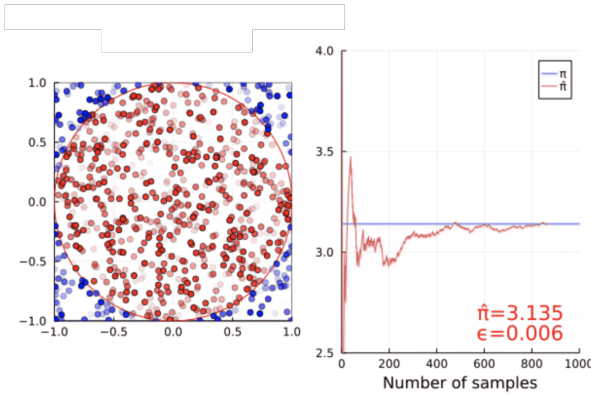


Figure 2: Illustrating how volume estimation can help determine the value of π. On the left-hand side we show the sample points, the blue points are outside the circle, and the red points are inside the circle. On the right-hand side, we see that as the number of samples increases, our approximation of π gets closer to the real value of π.

Now we provide the high-level idea of computing the number of linear extensions of a poset using volume estimation. Stanley [10] showed that one can

generate a polytope from a poset. Fisikopoulos et al. [4] developed a software package named volesti to compute the volumes of polytopes efficiently. Chalkis et al. [2] showed that one can compute the number of linear extensions by computing the volumes of the polytope of the poset. The goal of this project is:

- Implementing Gaussian Hamiltonian Monte Carlo random walk for counting linear extensions.
- Implementing the general mass matrix in the gaussian cooling algorithm of the volesti package.

The Hamiltonian Monte Carlo method has three main steps:

1. Picking a random velocity
2. Travelling deterministic-ally to the level set surface of the Hamiltonian
3. Projecting down in configuration space

Here is the pseudo code of the Hamiltonian Monte Carlo random walk:

Algorithm 1 Hamiltonian Monte Carlo with reflections

Choose the traveling time $L \sim$ unif(0, 1)
Pick the momentum $p \sim \mathcal{N}(0, I_n)$
if If the flow $\tilde{\phi}_L(q^{(t)}, p)$ is defined and does not involve more than M reflections between $t = 0$ and L, and if $\tilde{\phi}_L(q^{(t)}, p) \in Q$ **then**
  Take $q^{(t+1)} = \tilde{\phi}_L^{(q)}(q^{(t)}, p)$
else
  $q^{(t+1)} = q^{(t)}$
end if

The main code snippet of the algorithm looks like the following:

```
template
<
    typename GenericPolytope
>
inline void apply(GenericPolytope const& P,
                  Point& p,
                  NT const& a_i,
                  unsigned int const& walk_length,
                  RandomNumberGenerator &rng)
{
    unsigned int n = P.dimension();
    NT T;

    for (auto j=0u; j<walk_length; ++j)
    {
```

```
            T = rng.sample_urdist() * _Len;
            _v = GetDirection<Point>::apply(n, rng, false);
            Point p0 = _p;
            int it = 0;
            while (it < _rho)
            {
                auto pbpair = P.trigonometric_positive_intersect(_p, _v,
                    _omega, _facet_prev);
                if (T <= pbpair.first) {
                    update_position(_p, _v, T, _omega);
                    break;
                }
                _lambda_prev = pbpair.first;
                T -= _lambda_prev;
                update_position(_p, _v, _lambda_prev, _omega);
                P.compute_reflection(_v, _p, pbpair.second);
                it++;
            }
            if (it == _rho){
                _p = p0;
            }
        }
        p = _p;
    }
```

The cooling Gaussian method to compute volume first computes an annealing schedule. And then later uses that schedule to compute the volume. We implement the general mass matrix inside the schedule computation. The code snippet looks like this:

```
Point mean_point(n);
for(int i=0;i<n;i++)
{
        mean_point.set_coord(i, 0);
        NT sum = 0;
        for(int j=0;j<points.size();j++)
                sum += points[j][i];
        mean_point.set_coord(i, sum/n);
}
for(int i=0;i<n;i++)
{
        for(int j=0;j<n;j++)
        {
                covar(i, j) = 0;
                for(int k=0;k<points.size();k++)
                        covar(i, j) += (points[k][i]-mean_point[i])
                        *(points[k][j]-mean_point[j]);
```

```
                covar(i, j) = covar(i, j)/(n-1);
        }
}
```

5 Experimental result

We first run different well-known algorithms as well as the algorithms of the volesti packages and compare the running times in Table 1. We provide the running times of graphs having average degree 3. We denote volume estimation by VE in the acronyms. We have used 3 volume methods: cooling convex bodies (CB), cooling Gaussians (CG), and sequence of balls (SOB). We have used 2 rounding methods: minimum ellipsoid (ME) and SVD. We denote the algorithm by concatenation of acronyms, for example, VE-CB-ME denotes the volume estimation method that uses cooling convex bodies as the volume method and minimum ellipsoid as the rounding method. The walk length is $10 + d/10$, where d is the dimension of the polytope. And the error is equal to 0.1. We have run the algorithms of the AAAI 2018 paper [11]. We denote the telescoping product algorithm by TP-SLS. We denote the decomposition telescopic product algorithm by DTP-SLS. We denote the decomposition telescopic product using the Gibbs sampler by DTP-GLS. We denote the tootsie pop algorithm by TP-basic. We denote the exact dynamic programming by DP. We denote the variable elimination via inclusion-exclusion by VEIE. We also use the algorithms of IJCAI 2018 [12]. We denote relaxation tootsie pop, trivial relaxation tootsie pop, extension tootsie pop, and adaptive relaxation Monte Carlo by RTP, TRTP, ETP, and ARMC respectively. From the figures, we can see that there are several efficient algorithms in the volesti packages to compute linear extensions.

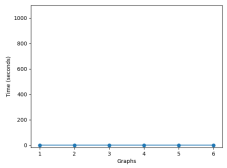
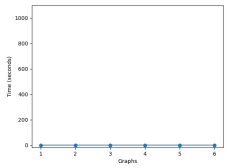
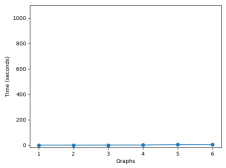
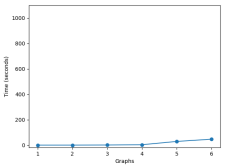
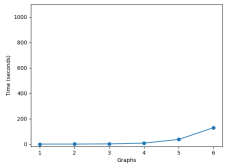
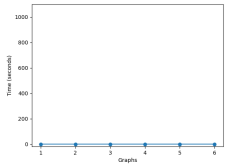
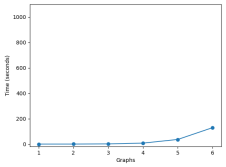
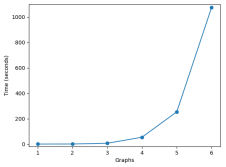
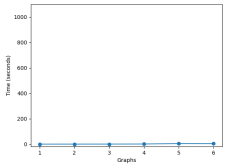
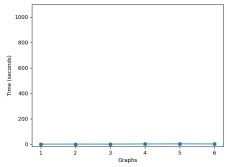
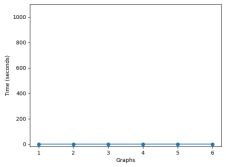
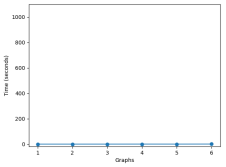
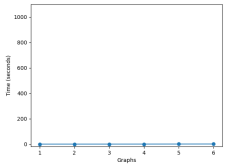
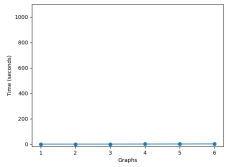
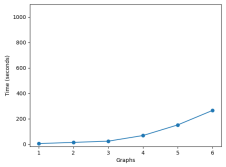
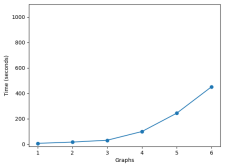
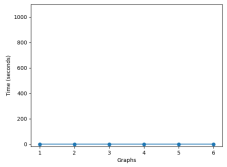
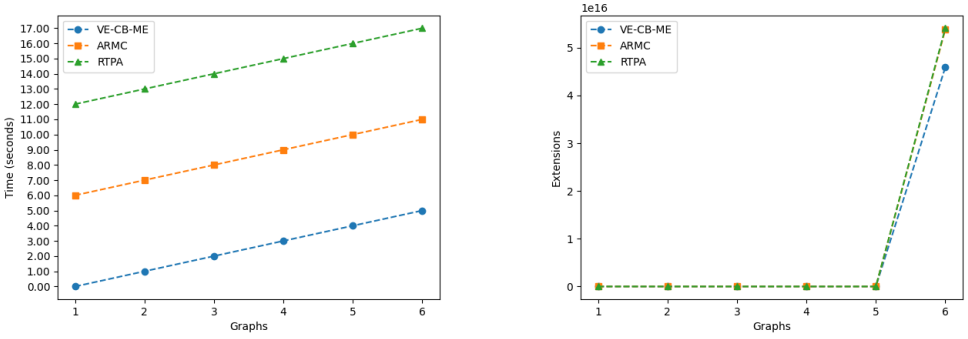
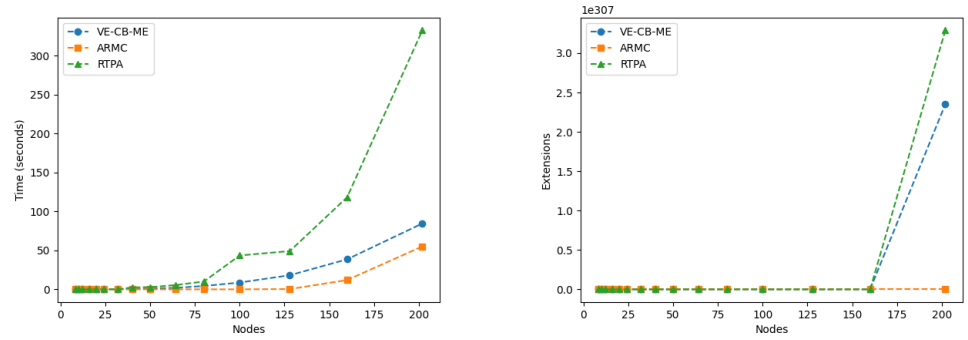
			
ARMC	DP	DTP-GLS	DTP-SLS
			
ETP	RTP	TP-basic	TP-SLS
			
TP	TRTP	VE-CB-ME	VE-CB-SVD
			
VE-CG-ME	VE-SG-SVD	VE-SOB-ME	VE-SOB-SVD
			
VEIE			

Table 1: Running times of graphs having average degree 3.

Since the illustration of Table 1 is a little congested we take out some of the algorithms that perform well from all those algorithms and make a separate comparison of those well-performing algorithms in Table 2. We can see the algorithm of the volesti package provides efficient solutions with good quality.



avg. degree 3



avg. degree 3

Time Extensions

Table 2: Running times and number of extensions. We denote the volume estimation method that uses cooling convex bodies as the volume method and minimum ellipsoid as the rounding method by VE-CB-ME. The walk length is $10 + d/10$, where d is the dimension of the polytope. And the error is equal to 0.1. We have compared the algorithm of the AAAI 2018 paper. We denote it by ARMC (adaptive relaxation Monte Carlo). We also compare the main algorithm of IJCAI 2018. We denote it by RTPA (relaxation tootsie pop algorithm).

Since we now know that there are several algorithms in the volesti package that can be used to approximate linear extensions, we have implemented the Gaussian Hamiltonian Monte Carlo algorithm in the volesti package framework and compared it with other algorithms in the package. We illustrate the results in Table **??**. We can see although our new method is providing correct results, however, it is slow.

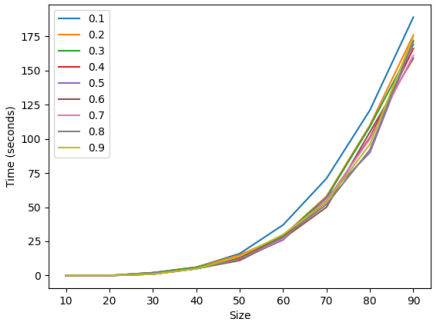
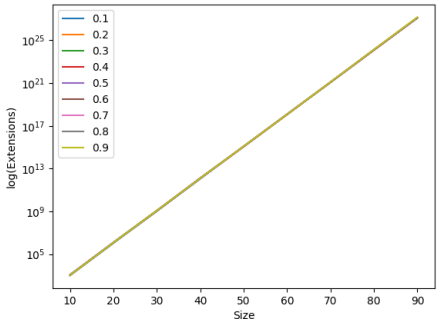
Hypercubes		
	Time	Volume

Table 3: Gaussian Hamiltonian Monte Carlo (HG) with different accuracy. Here, we have used a walk length equal to 1. As we can see, the runtime gets slightly better as the accuracy increases.

We now try tuning the parameters of the Hamiltonian Gaussian algorithm to reduce the running time. For example, there is a parameter ρ in the algorithm that controls the number of iterations of every step of the algorithm. We have tried different values of ρ as illustrated in Table 4. We can see the running time decreases as ρ decreases.

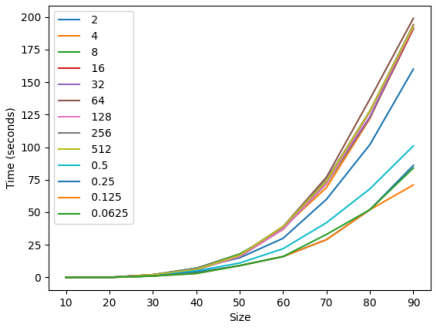
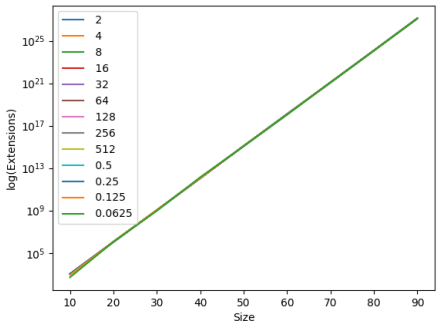
Hypercubes		
	Time	Volume

Table 4: Gaussian Hamiltonian Monte Carlo (HG) with different values of ρ. Here, we have used a walk length equal to 1. The diameter is set up to $2\sqrt{dim}$ (The default parameter). Here, we have changed the value of the multiplier of ρ by different factors (the default value was 100). As we can see, the runtime decreases as the factor decreases.

There is another parameter which is called the diameter, we have tried dif-

ferent values of diameter as shown in Table 5. We can see that the smaller the diameter, the better the running time is without that much change in volume.

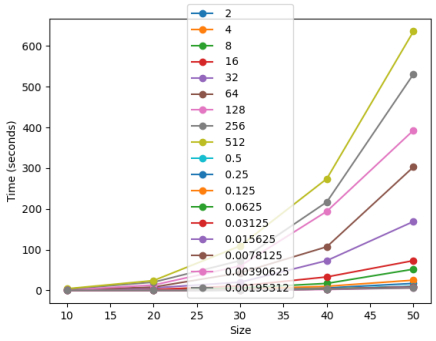
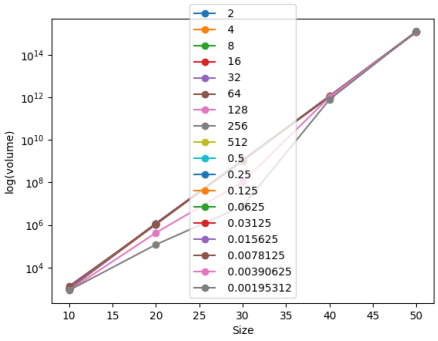
	Time	log(Volume)
Hyper-cubes		

Table 5: Here we have tried different diameters. The default diameter is $2 * \sqrt{dimension} * radius$. Here, we have set different factors of $\sqrt{dimension} * radius$ as diameter. We can see the smaller the factor, the better the run time without that much change in volume.

We have tried tuning several other parameters including the accuracy, tolerance, walk length, and the number of steps. However, those parameters do not have a large impact on running time. For example, we have shown the results of tuning the accuracy parameter in Table 6.

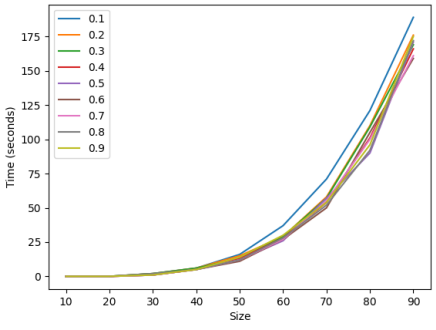
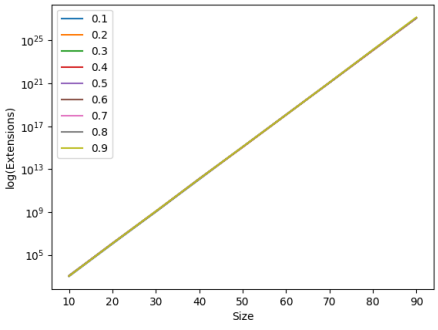
	Time	Volume
Hypercubes		

Table 6: Gaussian Hamiltonian Monte Carlo (HG) with different accuracy. Here, we have used a walk length equal to 1. As we can see, the runtime gets slightly better as the accuracy increases.

The window of the cooling Gaussian (CG) algorithm is another important parameter. The default window size is $6d^2 + 800$, where d is the dimension of the polytope. However, this default parameter was specified for the coordinate hit and run (CHNR) walk. Since CHNR takes more time to mix, and the time increases polynomially as the dimension increases, hence we specified that window parameter. However, the Gaussian Hamiltonian Monte Carlo walk (HG) can mix faster, hence we set the window parameter equal to 300 for this random walk. And HG performs better than the CHNR walk in the cooling Gaussian algorithm (CG+CHNR). We have shown the results in Table 7. The HG algorithm takes more time than the cooling ball (CB) algorithm, however, runs faster than the other algorithm.

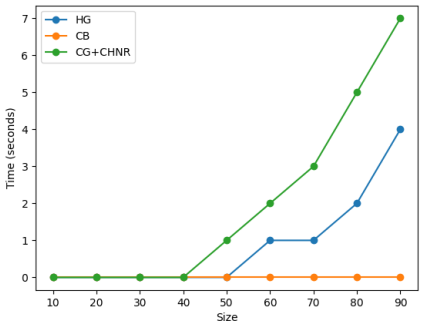
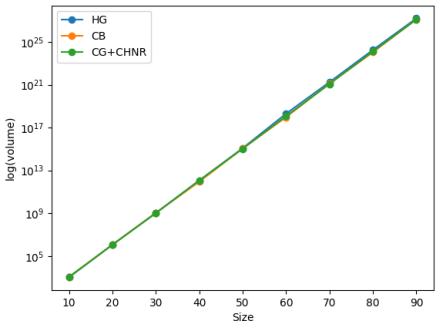
	Time	Volume
Hypercubes		

Table 7: Comparison of coordinate hit and run walk inside cooling Gaussian (CG+CHNR), the Gaussian Hamiltonian Monte Carlo walk (HG) and the cooling ball (CB) algorithm.

We run experiments to compare our implementation of the general mass matrix. We compare the Hamiltonian Gaussian walk with the general mass matrix (HG-GMM) to the Hamiltonian Gaussian walk with the SVD rounding (HG-SVD). For this experiment, we use the skinny polytope. The results are shown in Table 8. We can see that the accuracy degrades a little bit in HG-GMM.

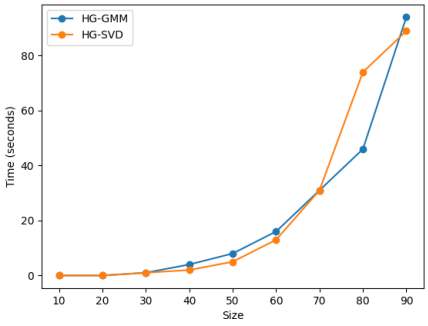
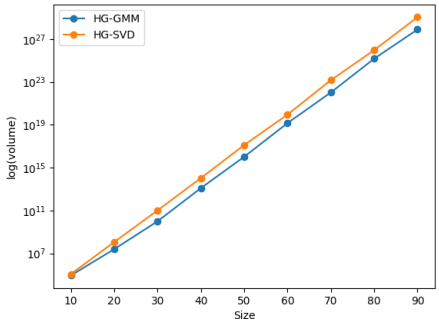
	Time	Volume
Skinny polytopes		

Table 8: Comparison of the Hamiltonian Gaussian walk with the general mass matrix (HG-GMM)to the Hamiltonian Gaussian walk with the SVD rounding (HG-SVD).

6 Conclusion

We have implemented the Gaussian Hamiltonian Monte Carlo algorithm for computing volume and integrated it to compute linear extensions. We have also implemented the general mass matrix. We have done extensive experiments on different datasets. By tuning different parameters we are able to improve the running time compared to the existing Gaussian cooling algorithm. However, our algorithm is still slower compared to the cooling ball technique. It remains an interesting future direction to further improve the running time Gaussian Hamiltonian Monte Carlo algorithm. Our implementation of the general mass matrix is slightly less accurate, making the algorithm more accurate with faster running time remains another future direction.

References

[1] Graham Brightwell and Peter Winkler. Counting linear extensions. *Order*, 8(3):225–242, 1991.

[2] Apostolos Chalkis and Vissarion Fisikopoulos. volesti: Volume approximation and sampling for convex polytopes in r. *arXiv preprint arXiv:2007.01578*, 2020.

[3] Karel De Loof, Hans De Meyer, and Bernard De Baets. Exploiting the lattice of ideals representation of a poset. *Fundamenta Informaticae*, 71(2-3):309–321, 2006.

[4] Vissarion Fisikopoulos and Apostolos Chalkis. Package 'volesti'. 2021.

[5] Kustaa Kangas, Teemu Hankala, Teppo Mikael Niinimäki, and Mikko Koivisto. Counting linear extensions of sparse posets. In *IJCAI*, pages 603–609, 2016.

[6] Alexander Karzanov and Leonid Khachiyan. On the conductance of order markov chains. *Order*, 8(1):7–15, 1991.

[7] Wing-Ning Li, Zhichun Xiao, and Gordon Beavers. On computing the number of topological orderings of a directed acyclic graph. *Congressus Numerantium*, 174:143–159, 2005.

[8] Christian Muise, J Christopher Beck, and Sheila A McIlraith. Optimal partial-order plan relaxation via maxsat. *Journal of Artificial Intelligence Research*, 57:113–149, 2016.

[9] Teppo Niinimäki, Pekka Parviainen, and Mikko Koivisto. Structure discovery in bayesian networks by sampling partial orders. *The Journal of Machine Learning Research*, 17(1):2002–2048, 2016.

[10] Richard P Stanley. Two poset polytopes. *Discrete & Computational Geometry*, 1(1):9–23, 1986.

[11] Topi Talvitie, Kustaa Kangas, Teppo Niinimäki, and Mikko Koivisto. Counting linear extensions in practice: Mcmc versus exponential monte carlo. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[12] Topi Talvitie, Kustaa Kangas, Teppo Niinimäki, and Mikko Koivisto. A scalable scheme for counting linear extensions. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*. International Joint Conferences on Artificial Intelligence, 2018.