Programming Paradigms Summative Report

Introduction

In this report I will outline my reasoning for my approach to this problem, using graphs from experiments of my own and known results to justify the choices I made.

Choice of Algorithm

There are two main categories of algorithms to solve the shortest vector problem:

1. Enumeration
2. Sieving

I used the Schnorr-Euchnerr variant of enumeration because for this coursework, minimising memory use is as important as minimising time taken, and sieving algorithms have exponential space complexity (Yasuda, 2020). In contrast, enumeration algorithms have polynomial space complexity (Nguyen, 2017). Also, whilst sieving has a better asymptotic time complexity, enumeration is competitive in practice (Nguyen, 2017) due to smaller constants.

Enumeration Steps

Enumeration algorithms can be divided into 3 steps:

1. Reduce basis
2. Get bound for shortest norm
3. Enumerate all points within the bound

Finding the bound

To get the bound, I used Minkowski's second theorem:

$$\lambda_1(L) \leq \sqrt{\gamma_n}\mathrm{vol}(L)^{1/n} \text{ for } 1 \leq r \leq n \text{ (Yasuda, 2020)}$$

Where $\mathrm{vol}(L) = \det(\mathbf{B})$ where $\mathbf{B}$ is a matrix such that its column vectors form the basis of the lattice.

To find the determinant of the matrix, since Gram-Schmidt needs to be computed for Schnorr-Euchnerr enumeration, I used the formula

$$det(\mathbf{B}) = \prod_{i=1}^{n} \|\mathbf{b}_i^*\|^2$$

Where $\mathbf{b}_i^*$ is the ith orthogonalized basis vector produced from Gram-Schmidt. Since γ_n is hard to obtain exactly, I used the bound of $\gamma_n \leq 1 + \frac{n}{4}$.

Reduction and Enumeration

I use LLL for reduction as it is fast even for high dimensions due to its polynomial time complexity (Yasuda, 2020). LLL works by repeating Gram-Schmidt and swapping basis until the Lovasz condition is satisfied for all basis. The Lovasz condition ensures the vectors are orthogonal enough, and corresponds to

$$\langle \mathbf{b}_k^*, \mathbf{b}_k^* \rangle > (\delta - \mu_{k,k-1}^2)\langle \mathbf{b}_{k-1}^*, \mathbf{b}_{k-1}^* \rangle$$

I set δ to 0.99 to yield more orthogonal vectors and improve stability (NTL, 2020). While increasing the time taken to compute LLL, it speeds up the enumeration step due to better vectors being obtained.

After experimentation on random uniform lattices generated for dimensions 1 to 40, I obtained the following graph for time taken:

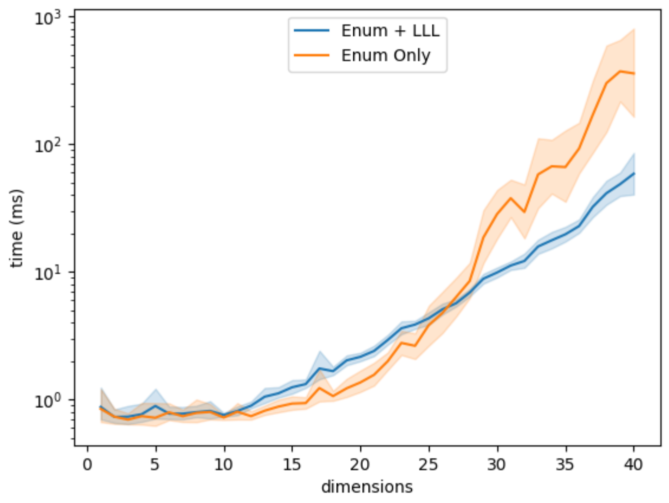

Figure 1

As shown by Figure 1, carrying out LLL only increases performance for dimensions greater than

27, this lead me to believe that I should only use LLL for dimensions greater than 27. However, this graph only considers uniform cases where the values of the basis vectors follow a uniform distribution. When considering hard cases for dimensions 2 to 11, I obtained the following results:

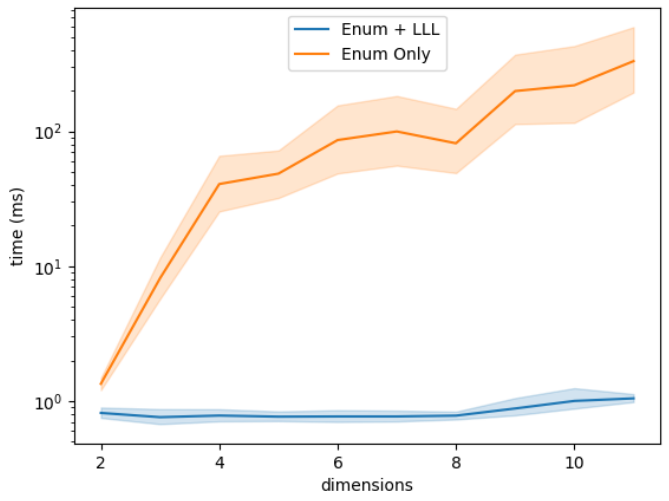

Figure 2

Hard cases are where the angle between the basis vectors is small (generated using latticegen -r with fplll). As Figure 2 shows, for such cases, LLL greatly reduces time taken even for small dimensions. Therefore, I use LLL for all dimensions.

Accuracy

Whilst I considered using solely LLL for small dimensions, even for small dimensions ($n < 5$) it is not guaranteed to give a correct solution (Polách, 2022), therefore I decided against this method. However, my final implementation gives the exact same answers as the answers given by fplll (to 5 decimal places), this was tested for hard cases with dimension less than 15 and uniform cases with dimensions less than 40.

Memory

I also measured the amount of memory used by LLL and enumeration and comparing that with merely using enumeration, and obtained the following graphs:

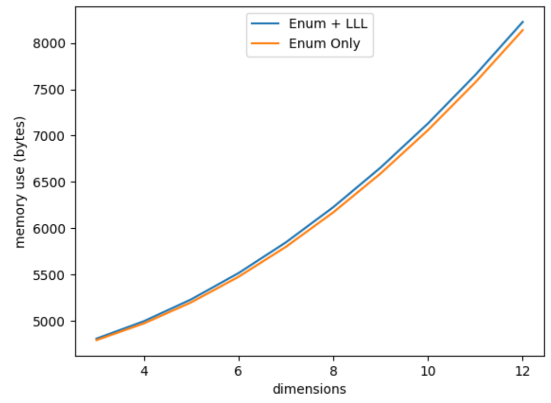

Figure 3: Memory use for hard cases

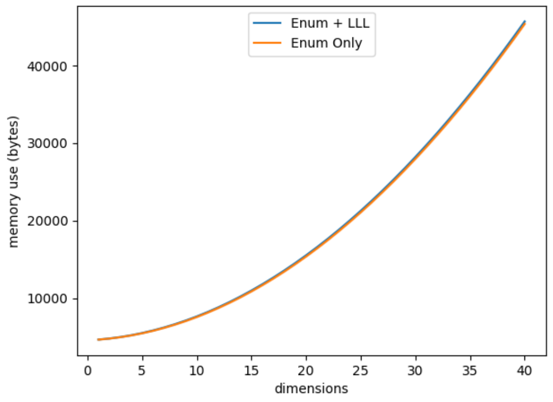

Figure 4: Memory use for easy cases

As Figures 3 and 4 show, adding LLL uses only slightly more memory, this is due to one extra array of size N, hence compared to the overall space needed to store the basis and Gram-Schmidt information which are of size N^2, this is not a major concern. This justifies my previous decision to use LLL as the performance increase it gives does not come with a great sacrifice in memory.

Low-Level Optimisations

Lower-level optimisations also had an impact on performance. Some of the ones I did were the following:
The first optimization I did was to minimize the number of malloc calls as dynamic memory allocation is relatively costly (Puaut, 2002). Hence, instead of using malloc inside of functions which are used frequently or loops, I made sure to do as much as possible in-place.
Another optimization was to use double for storing almost all my arrays. Since float was too small to get high accuracy, I use the next smallest data-type. This is to minimize memory usage and hence time whilst obtaining 100% accuracy, as shown by the graphs below:

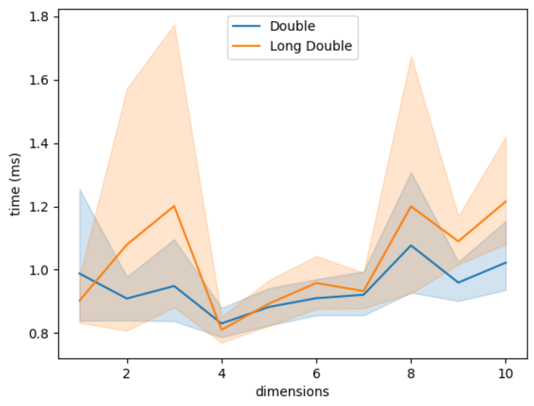

Figure 5: Time taken for double vs long double

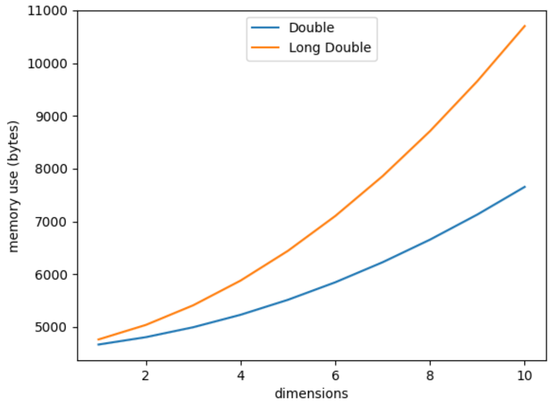

Figure 6: Memory use for double vs long double

Another optimisation was to save the computed values of the norm and store that alongside the Gram-Schmidt information. This is done as otherwise the norm would have to be recomputed.

References

Nguyen, P. (2017). Enumeration by pruning.

NTL. (2020). *Ntl: A library for doing number theory.* https://libntl.org/doc/LLL.cpp.html (accessed: 09.01.2023).

Polách, J. (2022). Lattice basis reduction using lll algorithm with application to algorithmic lattice problems.

Puaut, I. (2002). Real-time performance of dynamic memory allocation algorithms.

Yasuda, M. (2020). A survey of solving svp algorithms and recent strategies for solving the svp challenge. *International Symposium on Mathematics, Quantum Theory, and Cryptograph.*