Programming Paradigms

Seunghun Shin - qxck95

January 2024

1 Introduction

As science developed, the Shortest Vector Problem (SVP) has emerged in order to protect cryptographic systems from quantum computing attack. SVP is crucial foundation for secure lattice-based crpytogrpahic systems and post-quantum cryptography. The main focus of SVP is to efficiently find shortest vector with a given lattice. This report delves into run-time and memory requirements using C language.

2 Approach

Algorithm: The basic Schnorr–Euchner enumeration Schnorr and Euchner (1994)

Input: A basis $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$ of a lattice L and a radius R with $\lambda_1(L) \leq R$
Output: The shortest non-zero vector $\mathbf{s} = \sum_{i=1}^{n} v_i \mathbf{b}_i$ in L

Compute Gram–Schmidt information $\mu_{i,j}$ and $\|\mathbf{b}_i^*\|^2$ of $\mathbf{B}$
$(\rho_1, \ldots, \rho_{n+1}) = \mathbf{0}$, $(v_1, \ldots, v_n) = (1, 0, \ldots, 0)$, $(c_1, \ldots, c_n) = \mathbf{0}$, $(w_1, \ldots, w_n) = \mathbf{0}$
$k = 1$, `last_nonzero` $= 1$ // largest i for which $v_i \neq 0$
while `true` **do**
  $\rho_k \leftarrow \rho_{k+1} + (v_k - c_k)^2 \cdot \|\mathbf{b}_k^*\|^2$ // $\rho_k = \|\pi_k(\mathbf{s})\|^2$
  if $\rho_k \leq R^2$ **then**
    if $k = 1$ **then** $R^2 \leftarrow \rho_k$, $\mathbf{s} \leftarrow \sum_{i=1}^{n} v_i \mathbf{b}_i$; // update the squared radius
    else $k \leftarrow k-1$, $c_k \leftarrow -\sum_{i=k+1}^{n} \mu_{i,k} v_i$, $v_k \leftarrow \lfloor c_k \rceil$, $w_k \leftarrow 1$;
  else
    $k \leftarrow k + 1$ // going up the tree
    if $k = n + 1$ **then return s**;
    if $k \geq$ `last_nonzero` **then** `last_nonzero` $\leftarrow k$, $v_k \leftarrow v_k + 1$;
    else
      if $v_k > c_k$ **then** $v_k \leftarrow v_k - w_k$; **else** $v_k \leftarrow v_k + w_k$; // zig-zag search
      $w_k \leftarrow w_k + 1$
    end if
  end if
end while

Figure 1: SE algorithm pseudocode (Schnorr and Euchner 1994)

To tackle the coursework problem, I initially approached it by devising a Brute Force algorithm, leading me to formulate a structure where the range

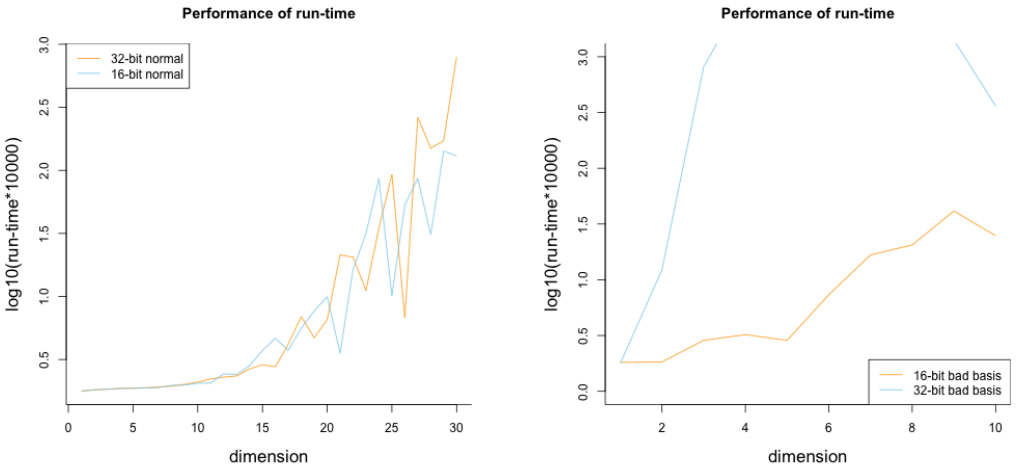

Figure 2: performance of runtime for 16-bit and 32 bit before LLL with normal and bad basis

is fixed and multiplied by 2 if we find a shortest vector. Reading advanced algorithm parts on a coursework instruction enabled me to research papers on online and find two basic enumeration algorithms: sieve and enumeration (Basic Schnorr-Euchner Enumeration) (Yasuda et al. 2021). There are several reasons I opted enumeration over sieve. Enumeration method has polynomial space, meaning that the memory required for enumeration algorithm will be $O(n^k)$ for some integer k. On the other hand, sieve method seeks for the shortest vector with exponential space, which implies $O(2^n)$. In general, when dimension n is low, polynomial space is more manageable compared to exponential space. Therefore, enumeration method is more pragmatic considering dimension of test will be less than 10. Furthermore, sieve is faster for higher dimension because it utilises the probabilistic approach to generate and filter through a large set of lattice points to find short vectors (Ajtai 1996). However, by enumeration's nature (deterministic step) as dimensions increase, the run-time performance decreases. The SE algorithm implemented is 1. Here 'bad' basis implies that basis is almost parallel. The left panel of Figure 2 illustrates the run-time performance of both 32-bit and 16-bit computations across dimensions 1 to 30 when using a normal basis. It is evident that the 16-bit computations exhibit faster run-time. This speed advantage can be attributed to the fact that arithmetic operations on 32-bit numbers generally take longer to execute compared to 16-bit numbers. Although the difference in run-time between the 16-bit and 32-bit normal basis computations is relatively small, the right panel of Figure 2 displays a significant difference in performance between 16-bit and 32-bit

computations when using a 'bad' basis.

3 Optimization

```
INPUT
    a lattice basis b_1, b_2, ..., b_n in Z^m
    a parameter δ with 1/4 < δ < 1, most commonly δ = 3/4
PROCEDURE
    B* <- GramSchmidt({b_1, ..., b_n}) = {b_1*, ..., b_n*};  and do not normalize
    μ_{i,j} <- InnerProduct(b_i, b_j*)/InnerProduct(b_j*, b_j*);   using the most current values of b_i and b_j*
    k <- 2;
    while k <= n do
        for j from k-1 to 1 do
            if |μ_{k,j}| > 1/2 then
                b_k <- b_k - ⌊μ_{k,j}⌉b_j;
                Update B* and the related μ_{i,j}'s as needed.
                (The naive method is to recompute B* whenever b_i changes:
                 B* <- GramSchmidt({b_1, ..., b_n}) = {b_1*, ..., b_n*})
            end if
        end for
        if InnerProduct(b_k*, b_k*) > (δ - μ^2_{k,k-1}) InnerProduct(b_{k-1}*, b_{k-1}*) then
            k <- k + 1;
        else
            Swap b_k and  b_{k-1};
            Update B* and the related μ_{i,j}'s as needed.
            k <- max(k-1, 2);
        end if
    end while
    return B the LLL reduced basis of {b_1, ..., b_n}
OUTPUT
    the reduced basis b_1, b_2, ..., b_n in Z^n
```

Figure 3: LLL algorithm pseudocode (Hoffstein, Pipher, and Silverman 2008)

In terms of improving run-time, I chose the Lenstra-Lenstra-Lovász (LLL) over the Block Korkin-Zolotarev (BKZ) algorithm. LLL is advantageous in smaller dimension in an aspect of complexity which makes more efficient and faster for smaller dimension. Whereas, BKZ is computationally intensive for larger block sizes (Yasuda et al. 2021). In light of evaluating our coursework under dimension 10, it is more appropriate to implement LLL algorithm. I also chose δ as 0.75 because the smaller delta results in higher precision but could take longer to obtain reduced basis. Therefore, LLL is suitable under circumstance with smaller n.

The left panel of Figure (4) demonstrates that the run-time performance before applying the LLL (Lenstra-Lenstra-Lovász) algorithm to a normal basis is faster compared to the run-time after applying the LLL algorithm to the same normal basis. This difference in run-time arises because the application of the LLL algorithm to a normal basis is not particularly effective in terms of reducing computational time to make the basis nearly orthogonal.Conversely, in the case of a 32-bit 'bad' basis, applying the LLL algorithm results in a dramatic improvement. The LLL algorithm has a substantial positive impact on the computational efficiency of the 'bad' basis. In terms of memory requirements, I have organized three two-dimensional arrays: basis, orthogonal basis, and Gram-Schmidt information μ.To optimize memory usage I have implemented an approach where I update the originally existing orthogonal basis rather than creating a new one each time it needs updating. This has resulted in reduced memory allocation. For dimension 1 and 30, allocated memories are 8,784 bytes and 29,432 bytes respectively.[figure (5)].

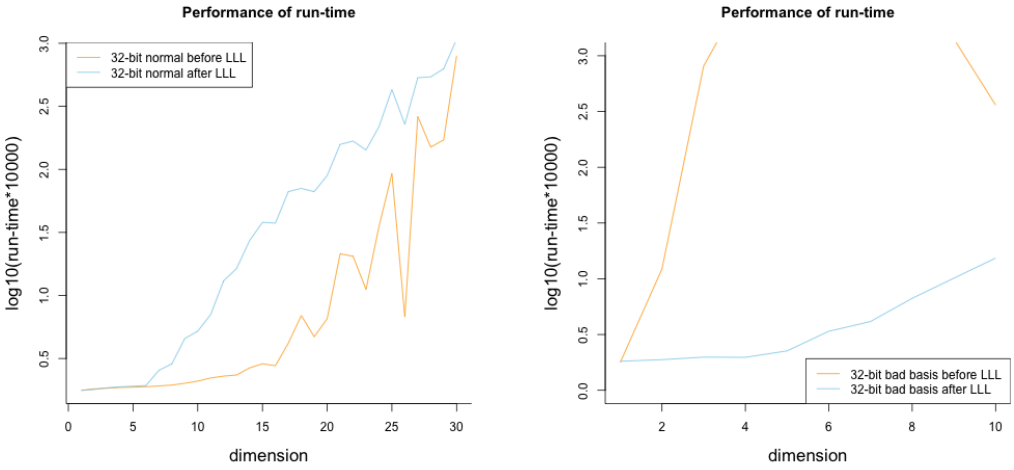

Figure 4: performance of runtime for 16-bit and 32 bit before LLL with normal and bad basis

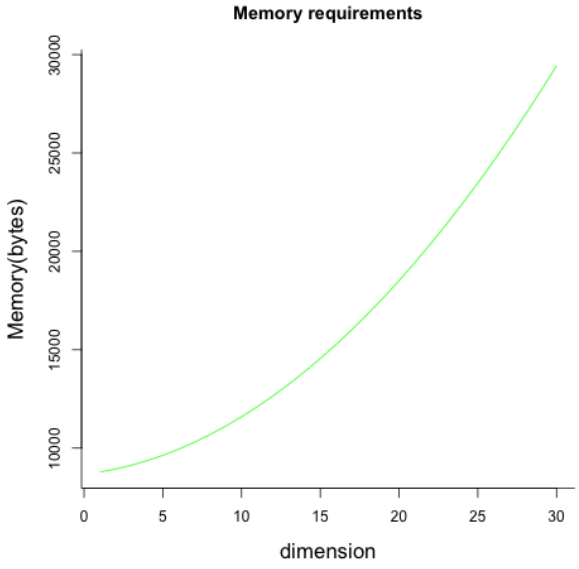

Figure 5: Memory usage

4 Error handling

Regarding input validation, I've implemented an algorithm that filters out specific cases. it also examines inputs for missing brackets, double bracket occurrences ([12]]), and wrong dimension formats ([1 2 3 4] [5 6] [7 8 9]). In terms of independence check, as it is hard to obtain determinant of given input basis in C, I have devised method to check if at least one of orthogonal basis elements is zero or not by introducing $\epsilon = 10^{-10}$ to avoid potential floating point errors.

References

Ajtai, Miklós (1996). "Generating Hard Instances of Lattice Problems". In: *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*. STOC '96. Philadelphia, Pennsylvania, USA: ACM, pp. 99–108. ISBN: 0-89791-785-5. DOI: 10.1145/237814.237838. URL: https://doi.org/10.1145/237814.237838.

Hoffstein, Jeffrey, Jill Pipher, and J.H. Silverman (2008). *An Introduction to Mathematical Cryptography*. Springer. ISBN: 978-0-387-77993-5.

Schnorr, C.P. and M. Euchner (1994). "Lattice basis reduction: Improved practical algorithms and solving subset sum problems". In: *Mathematical Programming* 66, pp. 181–199.

Yasuda, Masaya et al. (2021). "A Survey of Solving SVP Algorithms and Recent Strategies for Solving the SVP Challenge". In: *International Symposium on Mathematics, Quantum Theory, and Cryptography*. Singapore: Springer Singapore, pp. 189–207. ISBN: 978-981-15-5191-8.