SVP Report

hrwv51

I. Introduction

I explored algorithms beyond brute force for my project and identified two main categories: Sieving and Enumeration (with a lattice reduction algorithm). Sieving has exponential time complexity, while Enumeration is super-exponential ($O\left(n^{O(n)}\right)$) but often faster in practice for smaller dimensions. Enumeration also requires polynomial space, unlike Sieving's exponential space needs. I chose to implement Enumeration, specifically using the LLL [1] (Lenstra, Lenstra, Lovász) algorithm for lattice reduction. LLL has a time complexity of $O\left(d^5 n \log^3 B\right)$ where $d \leq n$ and B is the largest Euclidean norm of the basis vectors [2]. The alternative, BKZ (Block Korkine-Zolotarev), is slower than LLL but yields better-reduced lattices. Given my test cases won't be large, I prioritized speed over lattice quality.

When choosing which enumeration algorithm, I decided to use Schnorr–Euchner (SE) [3] enumeration as the other alternative I found was Kannan's algorithm which is recursive. SE has the advantage of being iterative, and in practice, is significantly faster than Kannan's algorithm.

Finally I used Minkowski's Theorem [4] to bound the initial search space, so that the algorithm would have a smaller initial search space, thus reducing the run time.

When writing the program, I followed pseudocode algorithms written in existing papers, which I have already referenced(SE [3], LLL [1]). Furthermore I tested various Gram-Schmidt [5] algorithms, in order to find the fastest solution. From the tested algorithms, the one referenced proved to be the fastest, and so I used it in my implementation.

For error checking, I ensure that the user has inputted a starting square bracket for each vector. Additionally I implemented a check for linear dependence, where the program will exit, if it is detected. Finally before writing the output, I check if its zero. If it is zero, an error must have occured during enumeration, and so it will print "Error." in console.

II. Analysis

A. Accuracy

I used fplll [6], which is a lattice reduction library, to generate test cases of varying dimensions, varying size (8, 16 and 32 bit) and varying types (knapsack and uniform). Fplll then generated the correct answers for each of test case. I then passed the generated case to `runme` and read the output, and compared it to the correct answer. I used 8285 cases to test the accuracy of my algorithm. Out of the 8285 cases, only 19 of the calculated answers were not within 1% of the true answer. Having said that, all of these cases were knapsack cases, and the output typically deviated by an average of 3%, with the highest deviance being 14.36%.

B. Speed

For speed, I generated a cases with larger dimensions. I then used `hyperfine` to measure the average time taken for each case, then averaged the outputs for each dimension. Something I did notice while optimising for speed was that in LLL, when you set δ closer to 1 for the Lovász condition, the code ran significantly faster in higher dimensions, with the tradeoff of being slower in smaller ones. As a result, I decided to leave $\delta = 0.85$, as I believe this provided the best balance for speed at both higher dimensions and lower dimensions.

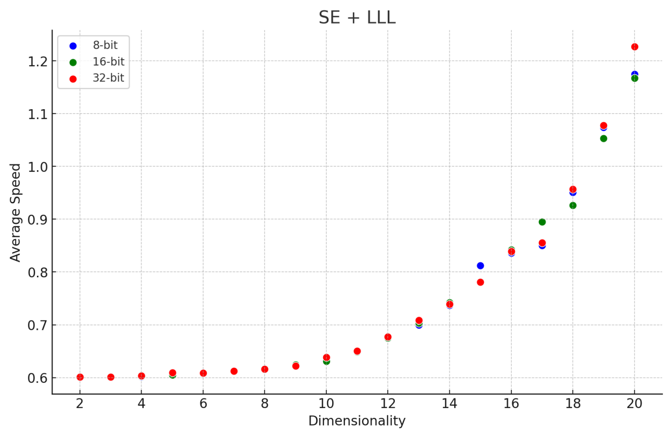

Figure 1: Uniform test cases. (Time in milliseconds)

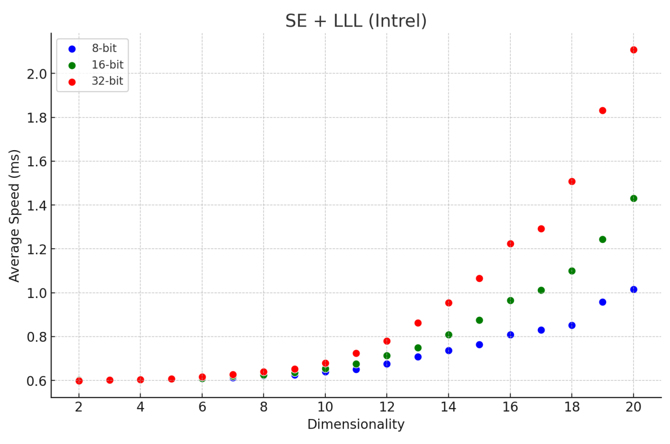

Figure 2: Knapsack test cases.

As you can see above, the Knapsack test cases are significantly slower, which is expected, as they are known to be harder than an average test case in the same dimension.

Figure 3 shows more uniform tests, ranging from dimensionality 30 to 40.

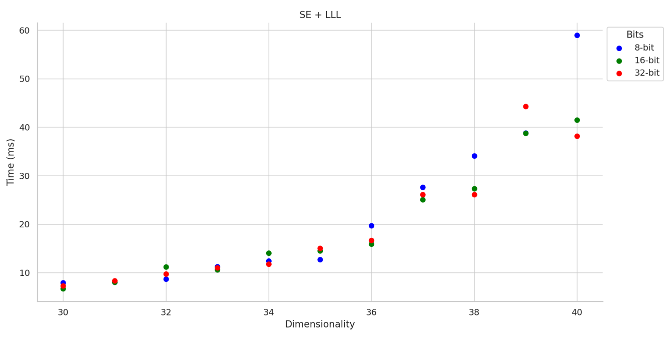

Figure 3: Uniform test cases.

These are the results for higher dimension knapsack test cases. Strangely, the 32 bit results appear to be very erratic. This could be due to the insufficient amount of test cases provided.

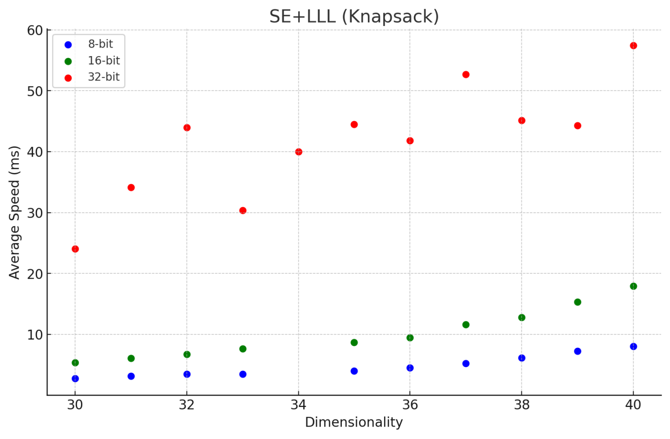

Figure 4: Knapsack test cases.

C. *Memory Usage*

To measure memory usage, I used valgrind [7], which outputs the total heap usage, and reported any leaks. From testing, the program did not have any memory leaks, and I have plotted the total heap usage below, to show the polynomial relationship between memory usage and dimensionality.

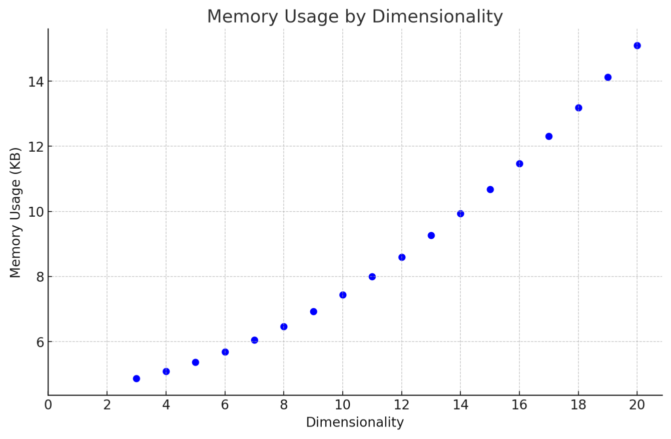

Figure 5: Memory usage.

When optimising for memory, I made sure to reuse as many temporary variables as much as possible, to streamline memory usage.

III. Conclusion

From the evidence provided, I believe my system works successfully; providing accurate results with relatively fast speeds and low memory usage. From the reading I have done, there is one further speed optimisation that I could have implemented, which would have saved me time by not recomputing Gram-Schmidt in the `else` of the Lovász condition.

Note: The system used to benchmark speed was an M2 Macbook Air 2022.

References

[1] J. M. Sanjay Bhattacherjee Julio Hernandez-Castro, “A Greedy Global Framework for LLL”. [Online]. Available: https://eprint.iacr.org/2023/261.pdf

[2] Anon, “Lenstra–Lenstra–Lovász lattice basis reduction algorithm”. [Online]. Available: https://en.wikipedia.org/wiki/Lenstra%E2%80%93Lenstra%E2%80%93Lov%C3%A1sz_lattice_basis_reduction_algorithm#cite_note-9

[3] M. Yasuda, “A Survey of Solving SVP Algorithms and Recent Strategies for Solving the SVP Challenge”, 2021, pp. 189–207. doi: 10.1007/978-981-15-5191-8_15.

[4] Anon, “Minkowski's theorem”. [Online]. Available: https://en.wikipedia.org/wiki/Minkowski%27s_theorem#:~:text=For%20n%20%3D%202%2C%20the%20theorem,in%20addition%20to%20the%20origin.

[5] L. C.-C. Satılmış H. Akleylek S., “Efficient Implementations of Sieving and Enumeration Algorithms for Lattice-Based Cryptography”. [Online]. Available: https://www.mdpi.com/2227-7390/9/14/1618

[6] T. F. development team, “fplll, a lattice reduction library, Version: 5.4.5”, 2023. [Online]. Available: https://github.com/fplll/fplll

[7] T. v. development team, “Valgrind”, 2023. [Online]. Available: https://valgrind.org/