SVP Report

hrwv51

I. Introduction

I began researching the possible algorithms to use other than brute force, and realised there were two main categories of competitive algorithms: Sieving and enumeration (with a lattice reduction algorithm). The time complexity for sieving is exponential, whilst for enumeration it's super-exponential. However, in practice, enumeration with lattice reduction tends to be faster for smaller dimensions. Additionally enumeration has the advantage of using polynomial space, where in contrast sieving is known to use exponential space. So I decided I would base my implementation on enumeration. For my lattice reduction algorithm I chose the LLL (Lenstra, Lenstra, Lovász) algorithm. It has a time complexity of $O(d^5 n \log^3 B)$ with $d \leq n$ and where B is the largest Euclidean norm from the vectors in the basis [1]. The alternative algorithm was BKZ (Block Korkine-Zolotarev) which runs slower than LLL, but produced better lattices. Once again, considering that the test cases won't be very large, speed would be more important than a better reduced lattice.

When choosing which enumeration algorithm, I decided to use Schnorr–Euchner (SE) 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 [2] 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. SE = [3] LLL = [4] Gram-Schmidt = [5]

II. Analysis

A. Accuracy

I used fplll [6], which is a lattice reduction library, to generate test cases of varying dimensions (1 to 10), varying size (8, 16 and 32 bit) and varying types (knapsack and uniform). Fplll then generated the correct answers for each of the generated basis. I then passed the generated basis to `runme` and read the output, and compared it to the corresponding fplll answer. All in all 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 used fplll to generate a wider variety of basis', still with the same varying bit size and types. I then used `hyperfine` to measure the performance for each case, and calculated the average time 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.

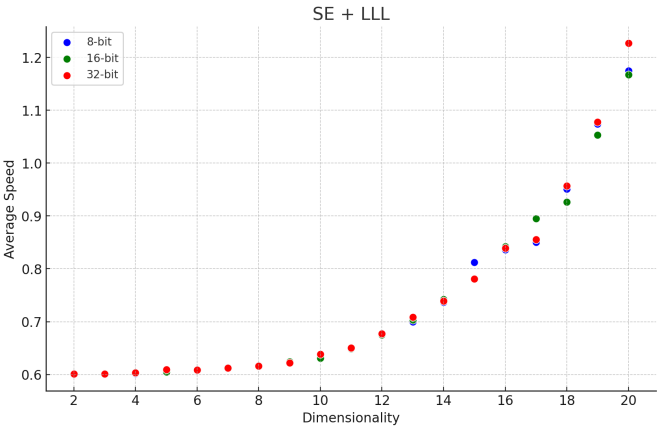

Figure 1: Uniform test cases.

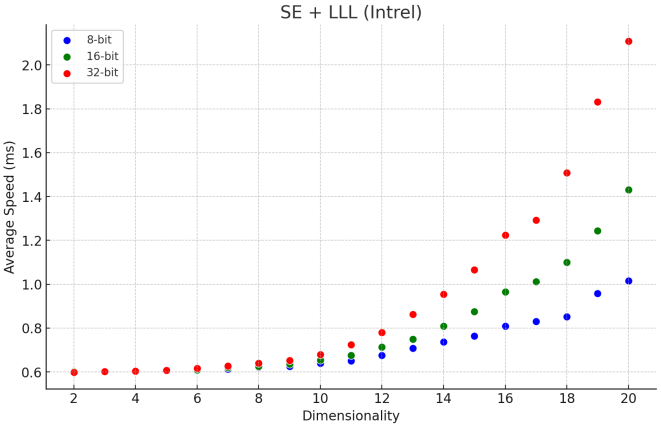

Figure 2: Knapsack test cases.

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.

Additionally I tested my implementation on higher dimensions. Figure 3 shows more uniform tests, ranging from dimensionality 30 to 50. It's clear from the trend, that the 8 bit 50d time is anomalous, as it does not fit with the trend.

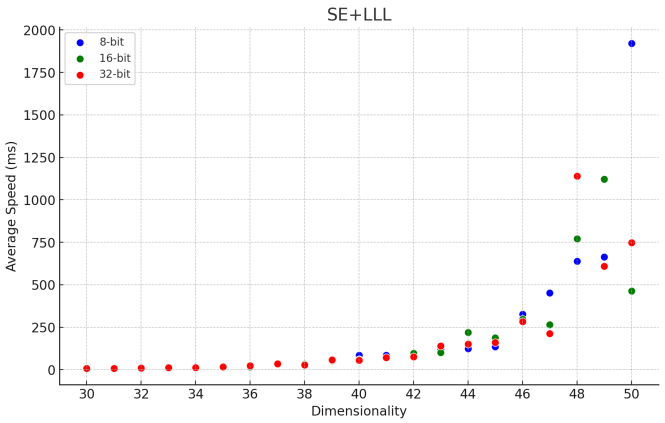

Figure 3: Uniform test cases.

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.

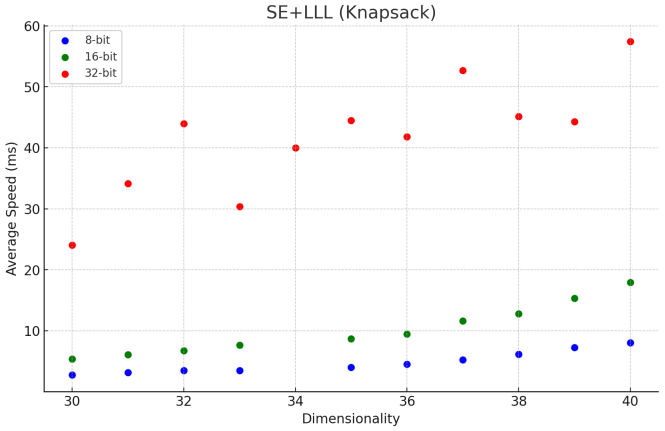

Figure 4: Knapsack test cases.

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.

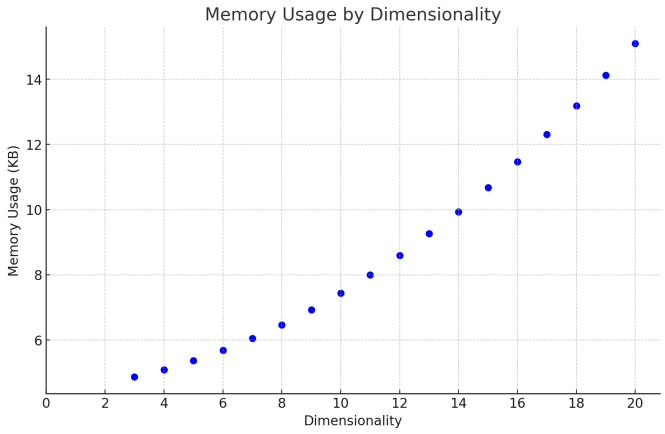

Figure 5: Memory usage.

When optimising for memory, I made sure to reuse as many temporary variables as 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] 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

[2] 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.

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

[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/