SOLVING THE SHORTEST VECTOR PROBLEM (SVP) IN C USING LENSTRA–LENSTRA–LOVÁSZ (LLL) LATTICE BASIS REDUCTION ALGORITHM

Ben Hughes

INTRODUCTION

The shortest vector problem is a cryptography problem to find the shortest possible vector than can be made from integer multiples of a given basis.

MY APPROACH

BRUTE FORCE APPROACH

After reading over the problem description my initial thought was to implement a simple brute force approach in 2 dimensions. My first implementation used 2 arrays to store the 2 basis vectors and it would use nested for loops to exhaustively search for the combination of the vectors with the shortest length. It would search within a certain vector space, defined by an argument $search\_resolution$.

Before extending the system for higher dimensions, I developed some code that would allow vectors to be input from the program arguments that simply read values from the $argv$.

I then extended my recursive search algorithm to work on n dimensional bases, however I had a problem with using for loops, as they would obviously not scale to multiple dimensions. I instead used a while loop over

$$(2 * search\_resolution + 1)\hat{}dimension$$

which would multiply each vector by a multiple in each search_ resolution direction, so search depth 2 would check -2 through to 2 multiplied by each vector and then sum them all together and work out the length. I realised there was a clear inefficiency as a lot of the partial vector sums could be reused.

I then used an iterative approach so that the same sums would not be repeated, as for example in a 10 dimensional basis if the search depth was 5 then 1 multiplied by the first vector and 1 multiplied by the second vector would be computed (2*5+1)**8 or 214358881 number of times, whereas with an iterative approach this sum would only be calculated once.

LENSTRA–LENSTRA–LOVÁSZ:

After implementing my brute force solutions I read into more complex lattice reduction algorithms to solve SVP and decided to implement Lenstra–Lenstra–Lovász (LLL) lattice basis reduction algorithm, which uses Gram Schmidt (GS) orthogonalization and size reduction to reduce a basis.

I found a pseudocode [1] application of LLL and tried to implement it in C. I first implemented matrix manipulation functions: inner product and GS coefficient, which I used to implement a function to orthogonalize a basis using GS orthogonalization. After implementing these I moved onto implementing the main LLL algorithm. My first implementation used 3 2D arrays as matrixes to store: the basis, the orthogonalized basis and the Gram Schmidt coefficients, however I then removed the 3rd matrix and instead and computed each coefficient only when it was needed, saving memory and time.

Overall system:

I then went back to my system that extracted the vectors from the command line and improved, making sure there are a square number of arguments, ensuring all vectors are opened and closed with square brackets and making sure arguments contain only numbers and single minus and dots.

I then finished by making a makefile with basic unit vectors as the tests, as well as some harder tests using the 40-dimensional bases from the examples site.

Analysis of algorithms

Brute force algorithm

my brute force algorithm takes an argument called $search\_resolution$, for each call of the search function it will call itself

$$search\_resolution * 2 + 1$$

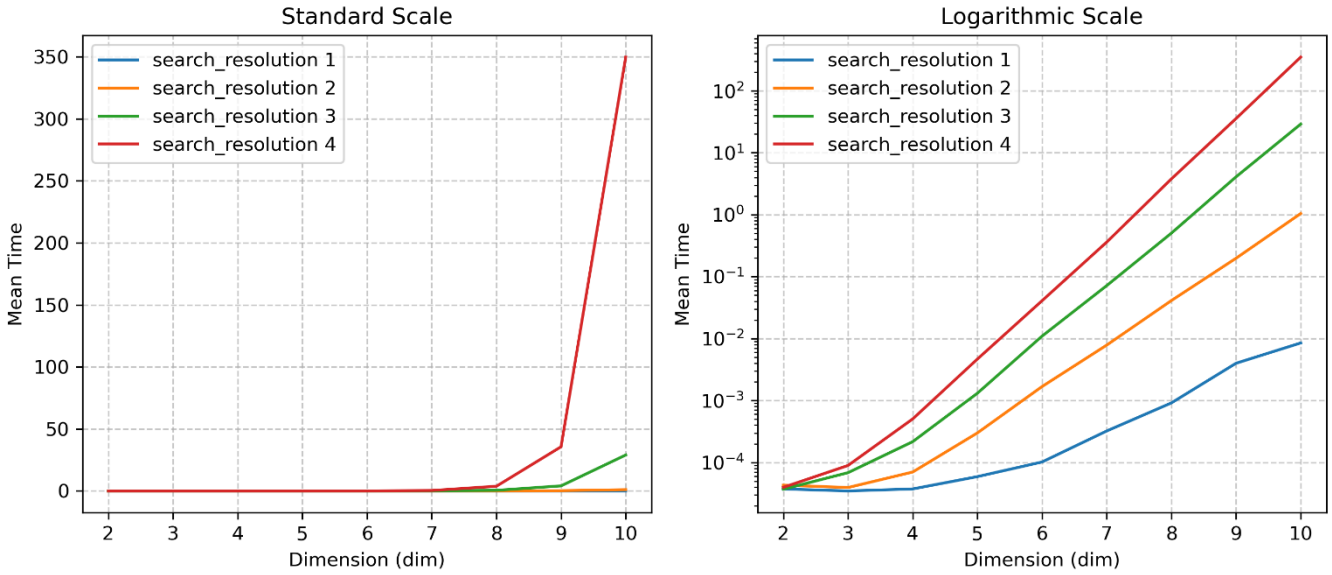
number of times, the depth of the function is the same as the dimension as it is called for each vector in the basis, so the complexity would be

$$O((search\_resolution * 2 + 1)\hat{}dimension)$$

while the recursive version does have a smaller running time the complexity is still the same. This exponential complexity can be seen in figure 1, as the logarithmic scale shows a straight line through the origin showing

$$log(time) = log(search\_resolution * 2 + 1) * dimension$$

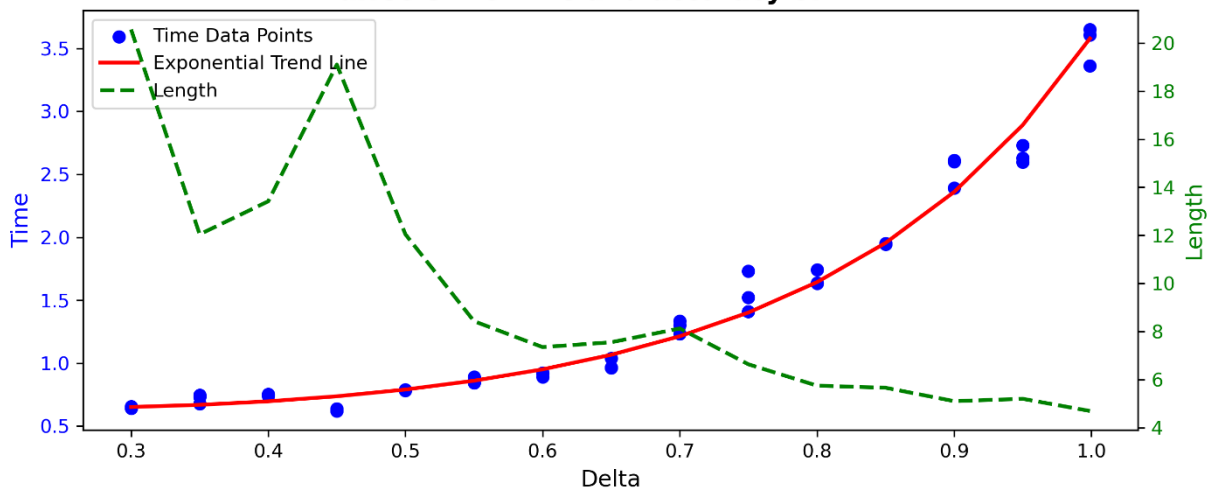
Figure 1. How Dimension and Search Resolution Affect the Running Time for the Brute Force Algorithm



LLL ALGORITHM:

My LLL algorithm takes an argument called delta, which is used in the Lovasz condition to compare whether vectors should swap in the LLL process, the value of delta determines the quality of the final solution as well as the time it takes to find said solution.

Figure 2. A plot to show how the value of delta affects the time and accuracy of LLL



I have chosen to use a delta value of 0.99 as it will give the optimal solution and from my testing never took more than 4 seconds to compute a value for a 40-dimensional basis and will always be solved In polynomial time [2] as $0.25 < delta < 1$

SUMMARY AND CONCLUSION

In the end I have decided to use the LLL algorithm. Overall, I am happy with my program, I believe it is of a high quality and am confident in its ability to effectively solve the SVP using the algorithms I have selected.

REFERENCES:

[1] Hoffstein, J., Pipher, J., & Silverman, J. H. (2008). An Introduction to Mathematical Cryptography (p. 411, Chapter 6.12: Lattice Reduction Algorithms). With corrections from errata. Silverman, J. (Retrieved 10th Jan 2024). "Introduction to Mathematical Cryptography Errata" (PDF). Brown University Mathematics Dept.

[2] Galbraith, S. D. (2018). Mathematics of Public Key Cryptography. Version 2.0. (Chapter 17.5: Complexity of LLL). Retrieved from https://www.math.auckland.ac.nz/~sgal018/crypto-book/main.pdf on January 10, 2024.

Word count excluding diagrams, images, titles, captions, tables, references, and graphs: 720