Report

Method

LLL Basis Reduction:

First, I reduce the basis vectors using Lenstra–Lenstra–Lovász basis reduction. This produces a new set of shorter and nearly orthogonal basis vectors, each an integer combination of the original basis vectors.

Enumeration:
I based my algorithm on the full enumeration algorithm, given in the appendix of (Zheng *et al.*, 2016).

I input the reduced basis A, and calculate the squared norms of the Gram-Schmidt (GS) orthogonalized vectors associated with A.

We aim to find lattice points within the radius of the current shortest vector that we've found. I define each lattice point by the combination of basis vectors used to form its associated position vector. The number of each basis vector that we've used is stored in an array x. I also initiate an array l, where:

$$l_i = \left(x_i + \sum_{j>i} x_j \mu_{j,i} \right)^2 \|GS\ vectors_i\|^2 \qquad i \in [0, dim-1],$$

l_i is equivalent to the squared length of the projection of the position vector onto the i^{th} GS vector. Since the GS vectors are orthogonal, $\sum_{i\geq 0} l_i = \ \|position\ vector\|^2$

Therefore, we aim to find a lattice point such that $\sum_{i\geq 0} l_i < (current\ shortest\ length)^2$. Due to symmetry, we can assume that x_{dim-1} is positive. x_{dim-1} also has the following upper bound:

$$l_{dim-1} = x_{dim-1}^2 \|GS\ vectors_{dim-1}\|^2 < (current\ shortest\ length)^2$$

Starting with $i = dim - 1$, we iterate backwards in the following way:

1. If $\sum_{j\geq i} l_j <\ (current\ shortest\ length)^2$, set x_{i-1} to the smallest integer such that $\sum_{j\geq i-1} l_j <\ (current\ shortest\ length)^2$
2. If no such integer exits, $x_i = x_i + 1$, and proceed to step 3. Else $i = i - 1$, return to step 1.
3. Check if $\sum_{j\geq i} l_j <\ (current\ shortest\ length)^2$.
   If yes, return to step 1. If not, go to step 4.
4. $i = i + 1, \quad x_i = x_i + 1$, return to step 3.

If we reach $i = 0$, and $\sum_{j\geq 0} l_j < (current\ shortest\ length)^2$, set $current\ shortest\ length = \sqrt{\sum_{j\geq 0} l_j}$

When we exceed the maximum x_{dim-1} we end the loop, and return $current\ shortest\ length.$

Results

I tested my program using the upper bound estimate $1.2 * \frac{\Gamma\left(\frac{n}{2}+1\right)^{\frac{1}{n}}}{\sqrt{\pi}} * \det(L)^{\frac{1}{n}}$ suggested in (*SVP Challenge*). The result of my program is almost always lower than this value, and very close if not. My program can also correctly identify the shortest vector from the D_n lattices.

Run-time efficiency

Time complexities:

- LLL-reduction - polynomial (roughly $O(n^6)$)
- Enumeration algorithm – $2^{O(n^2)}$
- Overall - $2^{O(n^2)}$.

To achieve maximum run-time efficiency, I implemented the following steps:

1. The LLL reduction algorithm greatly decreases the run-time for the lattice enumeration, whilst not significantly increasing the memory requirements.
2. I divided the lattice search into a separate thread for each possible value of x_{dim-1}. However, the maximum possible number of acceptable values for x_{dim-1} is 2^n, and 2^n threads would dramatically increase the memory requirements. Therefore, I capped the maximum number of threads open at one time at $\min\left(10, \frac{dim}{2}\right)$, so as not to increase the memory complexity of my program. Whilst this cap also means that the use of threads does not decrease the time complexity, in most cases the total number of iterations of my enumeration loop is below 2^{dim}, meaning that splitting these into up to 10 threads at a time will significantly decrease the run-time. I also implemented a running shortest length which all threads have access to.
3. I implemented a *start* variable in the *update_matrices* function, so that we only recalculate the rows of B which have changed since the previous call of *update_matrices*.

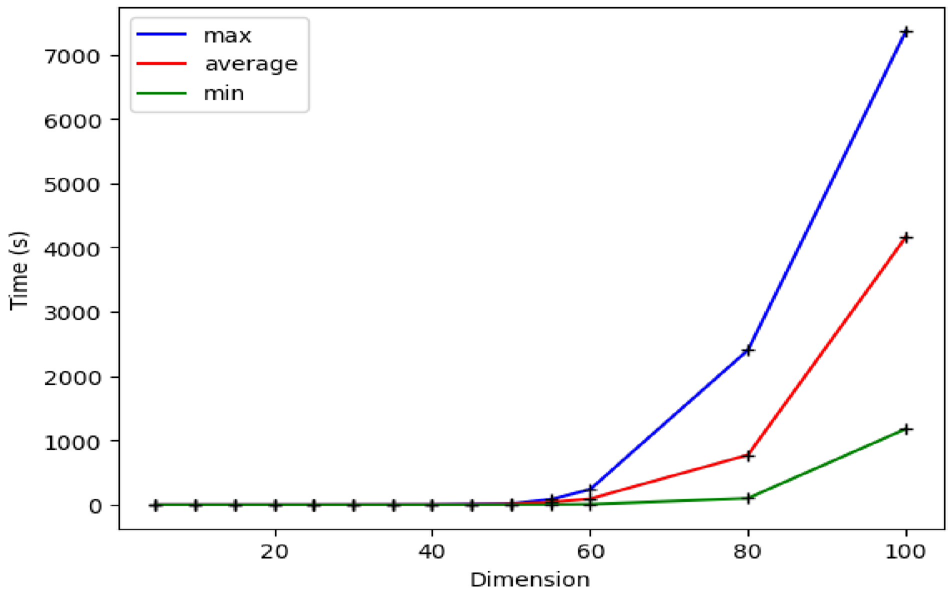

Figure 1: This graph shows the maximum, minimum, and mean average run-time out of 10 different random tests in each dimension. We can see an exponential increase in the average and maximum lines, and the minimum line would likely show the same trend for higher dimensions. More detailed results are shown in Table 1.

Dim	Average time (s)	Min time (s)	Max time (s)	Average lattice points	Min lattice points	Max lattice points	Average threads	Min threads	Max threads
5	0.00425	0.001	0.010	13.2	8	25	1.3	1	2
10	0.00852	0.003	0.013	120.8	18	248	1.5	1	2
15	0.00935	0.005	0.016	424.3	210	660	1.8	1	2
20	0.01675	0.007	0.038	5469.5	1683	11986	2.0	2	2
25	0.065	0.042	0.097	12839.2	5832	23840	2.4	2	3
30	0.414	0.233	0.715	301239.1	97227	581251	2.7	2	3
35	0.914	0.264	1.695	661728.0	188615	1443632	3.0	3	3
40	1.416	0.306	2.433	933911.7	156706	1728239	3.1	2	4
45	3.547	0.885	8.771	2355623.9	301655	6136824	3.3	2	4
50	8.512	2.177	15.231	5749386.9	1112268	10562747	3.2	3	4
55	38.969	2.324	83.858	24374456.2	1329891	54554170	3.8	3	5
60	66.669	6.437	237.485	41766591.4	3854322	88573926	3.6	2	5
80	180.172	101.759	2406.966	362059516.0	53970184	1121695287	4.0	3	5
100	3499.292	1180.126	7364.323	3510948430.8	1005437772	5269368837	4.7	4	5

Table 1: Results of 10 random tests in each dimension, showing the run-times, the number of lattice points searched through (i.e. the number of iterations of the enumeration loop), and the total number of threads required.

Memory efficiency

The space complexity my algorithm is $O(n^2)$

To reduce the total memory required by my program, I implemented the following steps:

1. The matrices A, B and Mu are shared between the LLL reduction and enumeration steps. This also decreases run-time as we don't have to re-compute B and Mu at the start of the enumeration.
2. Mu is stored as an 1D array of dimension $(dim - 1) * dim/2$ rather than as a matrix, as only the entries below the leading diagonal of the matrix are important.
3. In the *GramShmidt* function, I dynamically update B, so as not to have to store any values in an extra array.

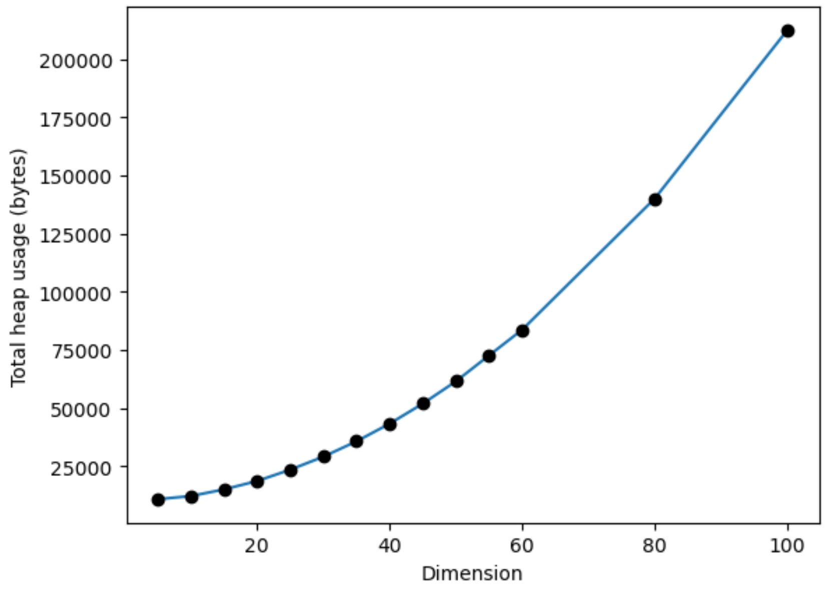

Figure 2: This graph shows the total heap memory allocated for different dimensional random input vectors (reported by valgrind –leak-check=full). It shows a quadratic increase with dimension, roughly following the curve $y = 20x^2 + 20x + 10500$. This is as expected with $8x^2$ coming from the allocation of rows in each of A and B, $4x^2 - 4x$ coming from the allocation of Mu, $8x$ coming from the allocation of GS_norms, and $8x$ coming from allocations of each of A and B. The $+10500$ comes from the memory overhead.

The main stack usage of my program comes from the threads, as x and l arrays are initiated in each thread. However, due to the cap on the number of threads open at one time, the maximum extra stack memory required by the threads is insignificant compared to the heap allocation.

Further considerations

When dealing with large numbers, the floating-point inaccuracy in C can cause incorrect results. To reduce this, in the *GramSchmidt* function, I normalise the vectors before computing the inner products. This helps to avoid doing multiplication with large numbers.

Note, when dealing with basis numbers > 10^{15}, the inaccuracies become too large, and the program will not return an accurate solution.

Bibliography:

SVP Challenge. Available at: https://www.latticechallenge.org/svp-challenge/index.php (Accessed: 11 January 2024).

Zheng, Z. *et al.* (2016) 'Orthogonalized Lattice Enumeration for Solving SVP'. Available at: https://eprint.iacr.org/2016/950 (Accessed: 10 January 2024).