Systems Programming Report

Jude Waide

1 Design Choices

1.1 Algorithmic Choices

To To solve SVP, the Kannan–Fincke– Pohst enumeration algorithm (KFP) [1] was used. This algorithm was chose as it is always guaranteed to provide an accurate answer unlike pruning techniques. Whilst pruning techniques have a much faster running time, they are not always guaranteed to find the answer which is a requirement in this instance.

KFP however is not enough by itself, as with high dimensional matrices the search space quickly becomes far too big to be solved in a reasonable time. This is especially the case with intrel matrices – the worst in which there are ones along the diagonals, a wide range of values in the 1^{st} element of each column, and zeroes everywhere else. To tackle this problem, the LLL algorithm [2] (polynomial running time) was used as a pre-processing step, which provides a reduced basis (ones who's vectors are more orthogonal and shorter) for the input lattice.

LLL can find the shortest by itself a lot of the time for smaller dimensions, the shortest vector appearing as one of the vectors in the reduced basis, however the success rate decreases as dimensions increase (success rate of $\tilde{9}0\%$ when n=10). For this reason, LLL must be followed by an enumeration technique to guarantee an answer. LLL however is still useful as by reducing the basis, a smaller initial bound for the smallest vector can be chosen, reducing the search space the KFP algorithm needs to look over.

With these 2 algorithms combined, even lattices with dimensions as high as 40 can be solved in a reasonable time. A small caveat is that if the shortest vector is within the reduced basis returned by LLL, the coefficients for the shortest vector returned by KFP will be all 0. This can be easily worked around however by checking the returned coefficients, taking the shortest basis vector of the reduced lattice in the case that they are all 0 and otherwise multiplying the coefficients by the reduced lattice to obtain the shortest vector.

1.2 Underlying Architecture Choices

Matrix and Vector classes were defined to manage memory automatically and make the process of writing the main algorithms easier. Move semantics were defined and used to avoid unnecessary copying, reducing memory consumption and time.

A standard dynamic array was chosen to store matrices and vectors as the size of the matrices isn't known at runtime, and a std::vector allocates more memory than is needed to allow for a size-flexible container which would be wasted memory in this instance as the size of the matrices won't change.

2 Performance Analysis

2.1 Procedure

To measure performance, the implementation was tested on random matrices of two types, intrel and regular (random values in every cell). To get the running time for the nth dimension, the average time taken to solve 5 random matrices of that size (generated using fyplll, bits set to 30) was taken. For memory, just a single matrix of that size was tested, using the program Dr. Memory.

2.2 Results and Analysis

2.2.1 Time

Table 1 shows that the implementation was a success, being able to solve svp for a worst-case n=40 in 16s. 1 visualises the results from the table, showing that the running-time for intrel matrices increases super-exponentially with respect to n, which matches the expected running time of KFP.

For regular matrices it's harder to the see relation, however the running time still increases at least polynomially with respect to n. n=35 is a bit of an outlier, which the running time decreasing from n=30. The only explanation for this is that the random matrices selected happened to be particularly easy ones for n-35 by chance.

The graph also confirms that intrel matrices are harder to solve than regular ones, which the intrel matrices taking 3-4 times longer to solve.

2.2.2 Memory

Figure 2 shows that the implementation was successful as it does not take up much memory, with only 70KB being used up for a regular matrix of size 40 - a small fraction of what modern computers have. The graph also shows that memory consumption is polynomial, which makes sense considering the number of values in a n by n matrix scales polynomially with respect to n.

Another trend shown is that intrel matricies take up less memory usage than normal ones. This was a surprising result considering no optimisations for sparse matrices was made, but it is possible that compiler optimisation made this occur.

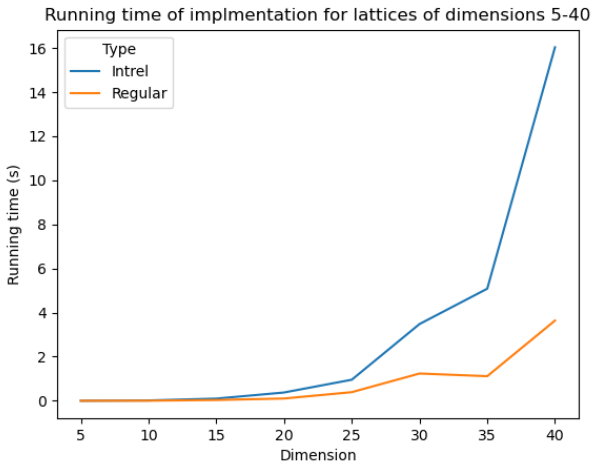

Fig. 1. Running time of algorithm against dimension

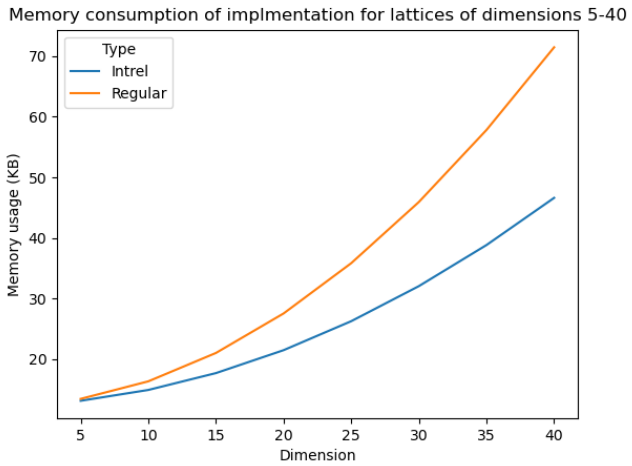

Fig. 2. Memory usage of algorithm against dimension

Type	Dimension	Time (s)	Memory (KB)
Intrel	40	16.0	46.6
Intrel	35	5.08	38.8
Intrel	30	3.48	32.1
Intrel	25	0.955	26.3
Intrel	20	0.376	21.5
Intrel	15	0.0991	17.7
Intrel	10	0.0151	14.9
Intrel	5	0.000808	13.2
Regular	40	3.64	71.5
Regular	35	1.12	57.8
Regular	30	1.24	45.9
Regular	25	0.391	35.8
Regular	20	0.103	27.6
Regular	15	0.0349	21.0
Regular	10	0.00384	16.4
Regular	5	0.000229	13.5

TABLE 1
Results table

REFERENCES

[1] Detrey, Jérémie & Hanrot, Guillaume & Pujol, Xavier & Stehlé, Damien. (2010). Accelerating Lattice Reduction with FPGAs. 124-143. 10.1007/978-3-642-14712-8_8.

[2] Deng, Xinyue. "An Introduction to Lenstra-Lenstra-Lovasz Lattice Basis Reduction Algorithm." (2016).