Balancing Act: Achieving Time and Memory Efficiency in SVP

Michal
Word Count: 750

Approach

My methodology was heavily influenced by my initial research into three of the most up-to-date methods of solving SVP:

Algorithm type	Time complexity	Space complexity
Enumeration	$n^{O(n)}$	$O(n^2)$
Sieving	$O(2^n)$	$2^{O(n)}$
Voronoi	$O(2^{2n})$	$O(2^n)$

Table 1: Time & Space complexities of varying types of Lattice-based algorithms

I considered all the options and while Sieving was conceptually the most intuitive, I found that Enumeration would be ideal for this task, due to its favourable time complexity.

Asymptotically, the time complexity of Enumeration is much worse than Sieving or Voronoi, however, empirical evidence suggests that for low-dimensions Enumeration outperforms them. Additionally, Enumeration has polynomial space complexity - much better than Sieving and Voronoi.

Although, Voronoi is a very cool way to solve this problem.

Lacking prior experience in C, I initially focused on getting a proof-of-concept working in Python.

I found there to be many basis-reduction algorithms, and while BKZ is most-commonly used, I struggled to implement it and instead implemented LLL.

Accuracy

A big worry of this assignment were floating point inaccuracies. Until testing, I didn't have an idea of how big of an issue they would be, hence I chose to use C's built-in `double`.

Arbitrarily, I set an accuracy threshold of $T = 5 \cdot 10^{-5}$.

This meant that if

$$|\text{Expected result} - \text{Actual result}| \leq T$$

then I would consider my result as correct.

Experimentally, I found this to be a better metric than percentage difference as it ensured correct results were closely aligned with the actual answer, and were unaffected by the result's magnitude.

For testing, I generated lattices using `latticegen` from the fplll library [1]. I treated this as my source of truth, since it is commonly referred to as the best lattice-based solver available. I made multiple bash & python scripts to automate this process, and focused my testing on uniform and knapsack-like lattices.

The reference I found most useful offered me a deeper understanding of the underlying theory and algorithms, and provided me with pseudocode for both:

LLL [2]

Algorithm: The basic LLL Lenstra et al. (1982)

Input: A basis $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$ of a lattice L, and a reduction parameter $\frac{1}{4} < \delta < 1$
Output: A δ-LLL-reduced basis $\mathbf{B}$ of L

Compute Gram–Schmidt information $\mu_{i,j}$ and $\|\mathbf{b}_i^*\|^2$ of the input basis $\mathbf{B}$
$k \leftarrow 2$
while $k \leq n$ **do**
  Size-reduce $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$ // At each k, we recursively change $\mathbf{b}_k \leftarrow \mathbf{b}_k - \lfloor \mu_{k,j} \rceil \mathbf{b}_j$ for $1 \leq j \leq k-1$ (e.g., see Galbraith 2012, Algorithm 24)
  if $(\mathbf{b}_{k-1}, \mathbf{b}_k)$ satisfies Lovász' condition **then**
    $k \leftarrow k + 1$
  else
    Swap $\mathbf{b}_k$ with $\mathbf{b}_{k-1}$, and update Gram–Schmidt information of $\mathbf{B}$
    $k \leftarrow \max(k-1, 2)$
  end if
end while

Figure 1: **Lenstra-Lenstra-Lovász** Basis-Reduction Algorithm pseudocode

... and **Schnorr-Euchner Enumeration** [2]

Algorithm: The basic Schnorr–Euchner enumeration Schnorr and Euchner (1994)

Input: A basis $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$ of a lattice L and a radius R with $\lambda_1(L) \leq R$
Output: The shortest non-zero vector $\mathbf{s} = \sum_{i=1}^{n} v_i \mathbf{b}_i$ in L

Compute Gram–Schmidt information $\mu_{i,j}$ and $\|\mathbf{b}_i^*\|^2$ of $\mathbf{B}$
$(\rho_1, \ldots, \rho_{n+1}) = \mathbf{0}$, $(v_1, \ldots, v_n) = (1, 0, \ldots, 0)$, $(c_1, \ldots, c_n) = \mathbf{0}$, $(w_1, \ldots, w_n) = \mathbf{0}$
$k = 1$, `last_nonzero` $= 1$ // largest i for which $v_i \neq 0$
while `true` **do**
  $\rho_k \leftarrow \rho_{k+1} + (v_k - c_k)^2 \cdot \|\mathbf{b}_k^*\|^2$ // $\rho_k = \|\pi_k(\mathbf{s})\|^2$
  if $\rho_k \leq R^2$ **then**
    if $k = 1$ **then** $R^2 \leftarrow \rho_k$, $\mathbf{s} \leftarrow \sum_{i=1}^{n} v_i \mathbf{b}_i$; // update the squared radius
    else $k \leftarrow k-1$, $c_k \leftarrow -\sum_{i=k+1}^{n} \mu_{i,k} v_i$, $v_k \leftarrow \lfloor c_k \rceil$, $w_k \leftarrow 1$;
  else
    $k \leftarrow k + 1$ // going up the tree
    if $k = n + 1$ **then return s**;
    if $k \geq$ `last_nonzero` **then** `last_nonzero` $\leftarrow k$, $v_k \leftarrow v_k + 1$;
    else
      if $v_k > c_k$ **then** $v_k \leftarrow v_k - w_k$; **else** $v_k \leftarrow v_k + w_k$; // zig-zag search
      $w_k \leftarrow w_k + 1$
    end if
  end if
end while

Figure 2: Basic **Schnorr-Euchner** enumeration algorithm pseudocode

Once I implemented both algorithms, I began testing different configurations, while varying δ.

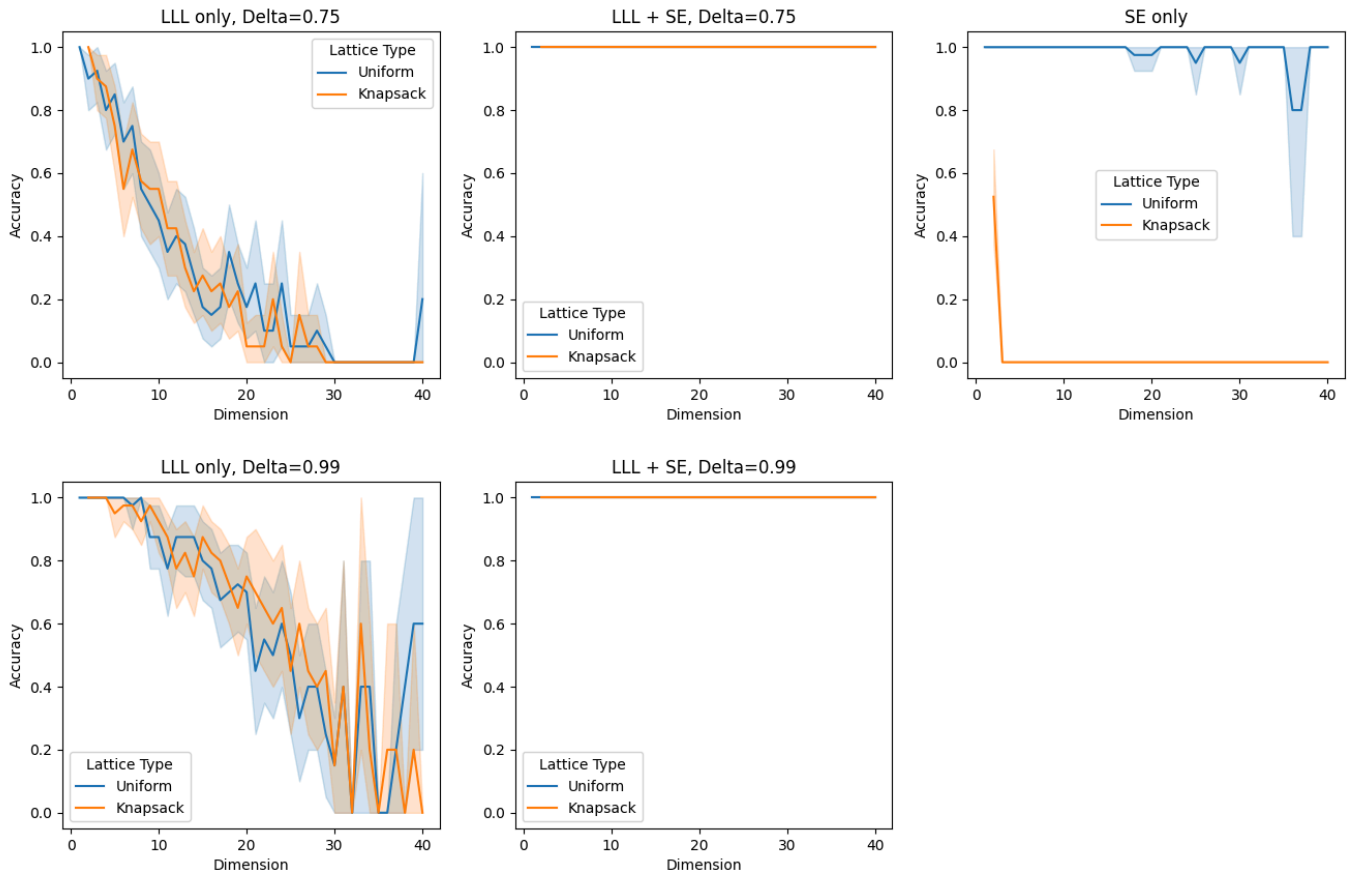


Figure 3: Accuracy vs. Dimension for various algorithm configurations

This highlighted that:

1. LLL and SE on their own had unsatisfactory accuracy
   - LLL simply gave approximations from its reduced basis which weren't always correct.
   - SE struggled with knapsack-like lattices (due to accumulating floating-point inaccuracies)
2. Increasing δ led to a higher accuracy. This aligns with other sources [3], and is due to a higher δ yielding a better basis reductionm, which ultimately leads to a better approximation.
3. LLL and SE combined exhibit superior performance.

To investigate whether it was necessary to use `long double` instead of `double`, I tested my LLL+SE implementation with both.

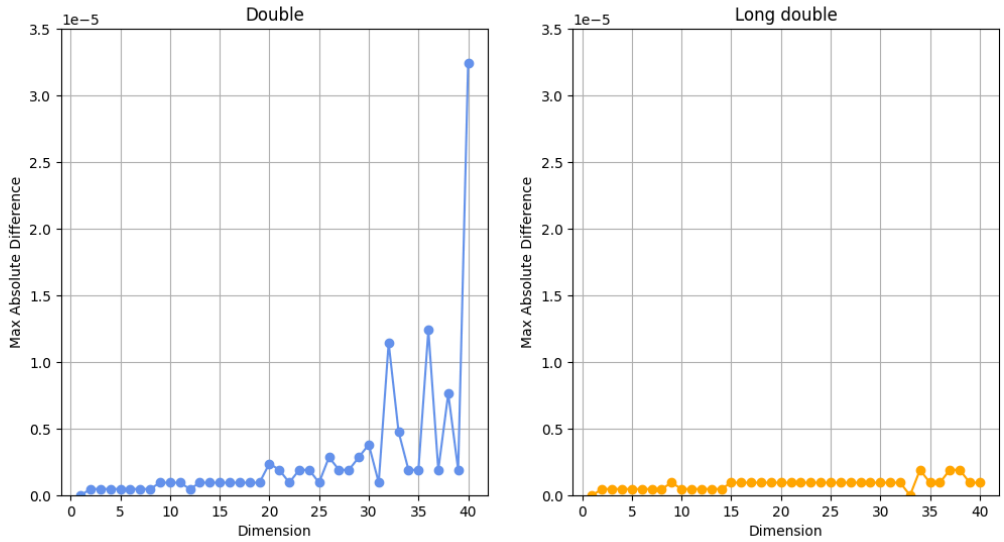
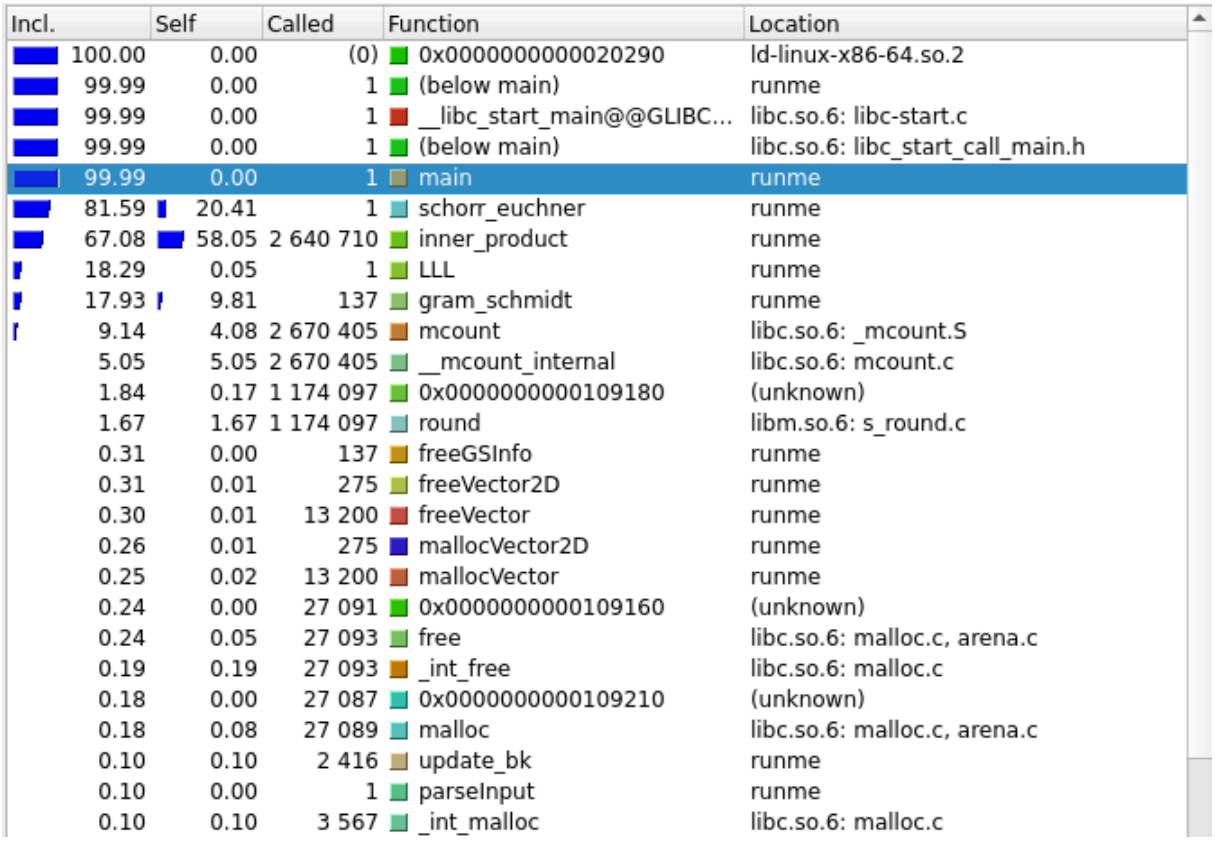


Figure 4: Maximum Absolute Difference vs. Dimension when using double and long double in LLL+SE, δ=0.99

Based on these findings, and considering the assignment's requirements, I concluded it was unnecessary as the accuracy stays within the tolerance T. Furthermore, I could not justify the extra memory and computation time needed for `long double` in this context.

Time

Upon running `valgrind`'s `callgrind` and feeding the result into `kcachegrind`, it became evident where optimisations would be most beneficial.



Incl.	Self	Called	Function	Location
100.00	0.00	(0)	0x0000000000020290	ld-linux-x86-64.so.2
99.99	0.00	1	(below main)	runme
99.99	0.00	1	__libc_start_main@@GLIBC...	libc.so.6: libc-start.c
99.99	0.00	1	(below main)	libc.so.6: libc_start_call_main.h
99.99	0.00	1	main	runme
81.59	20.41	1	schorr_euchner	runme
67.08	58.05	2 640 710	inner_product	runme
18.29	0.05	1	LLL	runme
17.93	9.81	137	gram_schmidt	runme
9.14	4.08	2 670 405	mcount	libc.so.6: _mcount.S
5.05	5.05	2 670 405	__mcount_internal	libc.so.6: mcount.c
1.84	0.17	1 174 097	0x0000000000109180	(unknown)
1.67	1.67	1 174 097	round	libm.so.6: s_round.c
0.31	0.00	137	freeGSInfo	runme
0.31	0.01	275	freeVector2D	runme
0.30	0.01	13 200	freeVector	runme
0.26	0.01	275	mallocVector2D	runme
0.25	0.02	13 200	mallocVector	runme
0.24	0.00	27 091	0x0000000000109160	(unknown)
0.24	0.05	27 093	free	libc.so.6: malloc.c, arena.c
0.19	0.19	27 093	_int_free	libc.so.6: malloc.c
0.18	0.00	27 087	0x0000000000109210	(unknown)
0.18	0.08	27 089	malloc	libc.so.6: malloc.c, arena.c
0.10	0.10	2 416	update_bk	runme
0.10	0.00	1	parseInput	runme
0.10	0.10	3 567	_int_malloc	libc.so.6: malloc.c

Figure 5: Snippet of function call summary provided by kcachegrind

My `schnorr_euchner`, `lll`, and `gram_schmidt` all relied on calculating millions of `inner_products`.

```
double inner_product(const Vector v1, const Vector v2, const int dim) {
    double total = 0;
    for (int i = 0; i < dim; i++) {
        total += v1[i] * v2[i];
    }
    return total;
}
```

Code sample 1: My implentation of the Euclidean Inner Product

The only optimisation that stood out was potentially parallelising using multiple threads. This however would only be effective on higher dimensions due to overhead.

I realised that by memoising/precalculating the inner products I could drastically reduce the number of calls to `inner_product`, thereby decreasing the number of calculations.

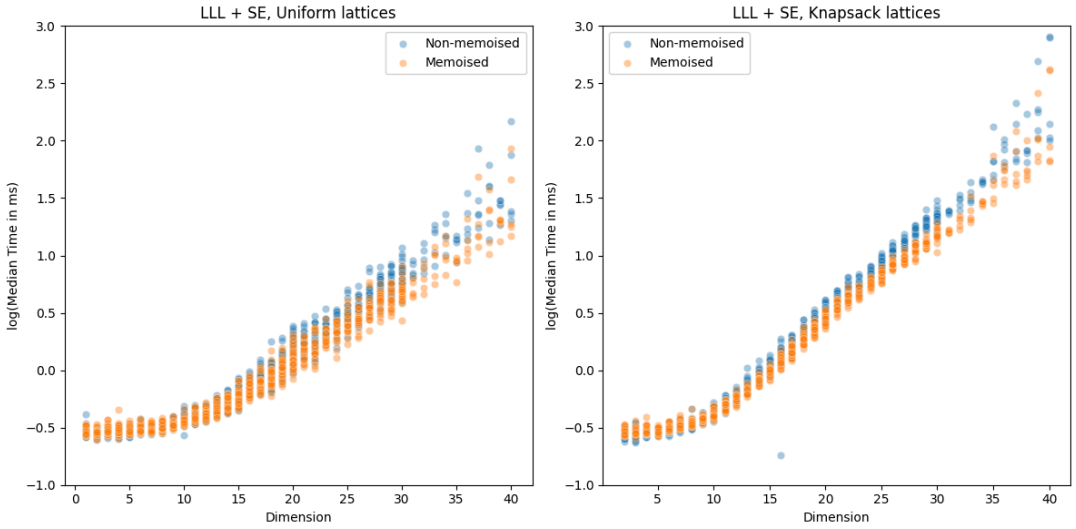


Figure 6: Effects of memoisation on median run-time of LLL + SE, δ=0.99

While the graph isn't perfect, it shows that memoising has a clear impact on time - this is amplified by the y-axis being logarithmic.

I was intrigued by the δ parameter, and decided to investigate more.

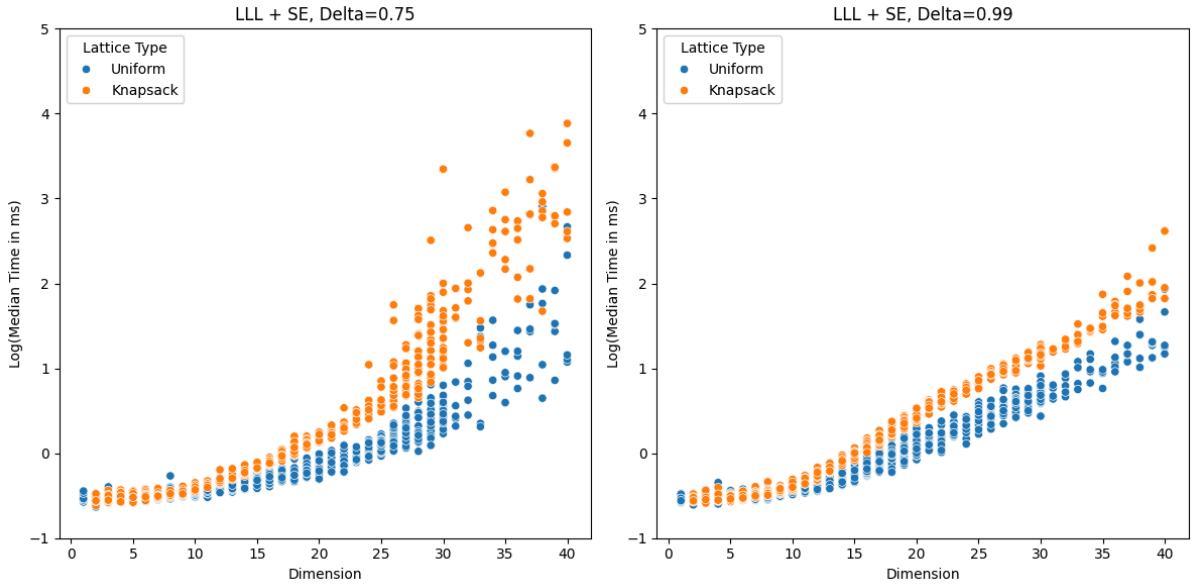


Figure 7: Effects of changing LLL's δ value on median run-time of LLL + SE

What I found was that for both uniform and knapsack lattices, a higher delta resulted in less variance of run-time, evident by the points being less scattered in Figure 7. It is important to note that for dimension 15-30, the run-time did increase due to overheard from computing LLL.

Memory

Another key aspect of optimisation was memory usage. For this, I used `valgrind` (tools: `massif` and `dhat`) which provided valuable insights that helped me address potential memory leaks and segmentation faults. The `memusage` tool was also useful as it gave me a distribution of memory block sizes, and from this, I could pinpoint inefficiencies in my data structures.

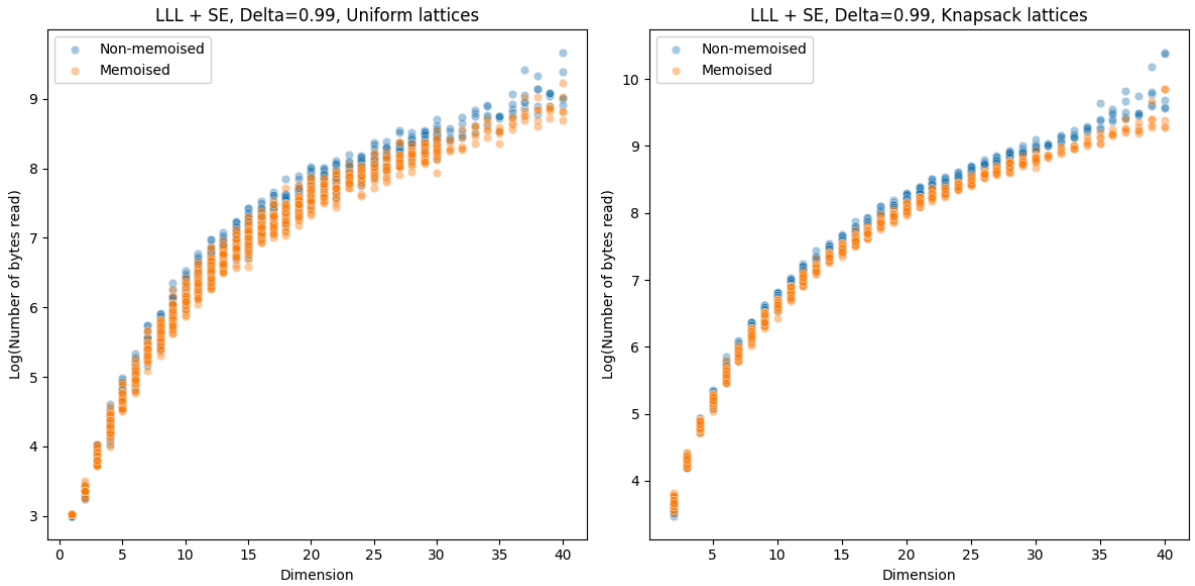


Figure 8: Effects of memoisation on the number of bytes read from memory

Readability

The final aspect I chose to prioritise was readability.

Overall Success

Overall, I believe my implementation is quite fast, and accurate to a high number of dimensions. In the future, I am planning to attempt this problem again with a Domain-Specific Language and with more advanced methods to improve upon the performance.

Bibliography

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

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

[3] N. (. Landsman, G. Lord, and G. Heckman, “Reduction of Matrices and Lattices”. [Online]. Available: https://www.math.ru.nl/magma/text536.html

1. Struct to typedef
2. malloc optimisation