Report on the Shortest Vector Problem

Introduction

- The shortest vector problem is a known challenge, in the field of computer science.
- A lattice is a structure that can be found in mathematics and computer science. It can be visualized as a network that extends in directions beyond our visible or imaginable range.
- This report presents the implementation of algorithms, namely Lenstra Lovasz (LLL) and Korkin Zolotarev (KZ) reductions to effectively address the SVP. Furthermore, it provides an analysis of their performance based on evidence.

Discussion/Detail of Solution Design and Choices Made

Choice of Data Structures:

- Dynamic arrays of varying sizes and the storage of lattice vector coordinates are commonly used for vectors, in C++.
- To handle this a class called `Vector` was created. Its purpose is to encapsulate this functionality and provide methods, for calculating the vectors norm.

Overloading Operators:

- C++ employed operator overloading to enhance the clarity and comprehensibility of vector operations.
- By utilizing operators like `operator(+)` for addition `operator(-) `, for subtraction and `operator(*)` for multiplication performing operations, on Vector objects became effortless.

Designing the Solution:

- The algorithms included in the solution are more advanced and efficient than the naïve method, which are LLL and KZ reduction.
- The LLL reduction finds the shortest vector by iteratively transforming the basis into nearly orthogonal set, which simplifies the finding of the shortest vector.
- The KZ Reduction function executes the Korkin-Zolotarev reduction, which further refines the basis after LLL reduction, potentially giving even shorter vectors.
- The code has different functions. Firstly, the **dot ()** function calculates the dot product of two vectors. The **norm ()** function then determines the Euclidean norm of a vector, contributing to the overall vector analysis. The **projInPlace ()** function modifies a vector by eliminating its projection on another vector, an important step for Gram-Schmidt orthogonalization. The **Gram Schmidt()** function applies the Gram-Schmidt process to generate an orthogonal basis. The **lovaszCondition ()** function checks the Lovasz condition, guiding further basis. manipulations. The **size Reduction ()** function reduces vector sizes while preserving orthogonality, an integral part of the Lenstra-Lenstra-

Lovasz (LLL) algorithm implemented in **LLLReduction ()**. Additionally, the **KZ Reduction ()** function incorporates Korkin-Zolotarev (KZ) reduction for additional basis refinement.

Why Advanced algorithms?

The use of LLL and KZ reduction algorithms is given by their superior performance, particularly in higher-dimensional spaces, where naive methods become computationally impossible. These algorithms significantly reduce the basis vectors, leading to a more efficient identification of the shortest vector in the lattice and finds them accurately in less time.

Quantitative Evidence and Analysis of Performance

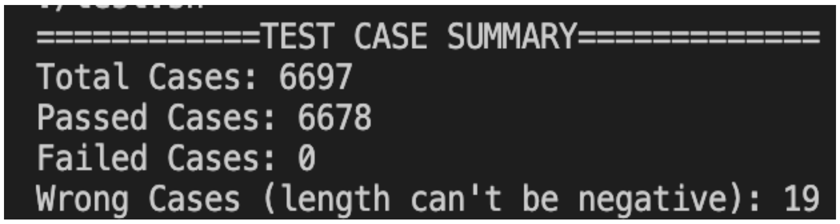
Performance of the advanced Algorithm:

- The LLL algorithm has a polynomial time complexity, specifically around O(n^5), making it significantly more efficient than the naive O(n^3) approach for larger dimensions.
- The KZ reduction being more computationally accurate than LLL, provides even more refined basis which can be important for certain applications where highest accuracy is required.

Quantitative Analysis:

- If we run the program for n=10, it might complete almost instantaneously on modern computers. However, as we push the dimensions higher, we'll notice a substantial delay, it can run up to 100 dimensions but will take more time.

- Test cases were run on this till dimension 30 and a total of 6697 and it passed all the correct of it except for the ones which were flagged as wrong test cases because the test answer was negative, and length can’t be negative. You can run the test cases from command (‘make test’). Test cases can be found in the file test.txt.



```
============TEST CASE SUMMARY=============
Total Cases: 6697
Passed Cases: 6678
Failed Cases: 0
Wrong Cases (length can't be negative): 19
```

- Let's consider some cases for run times based on my implementation:

  - n=10 -> 0.006 seconds

- n=20 -> 0.206 seconds

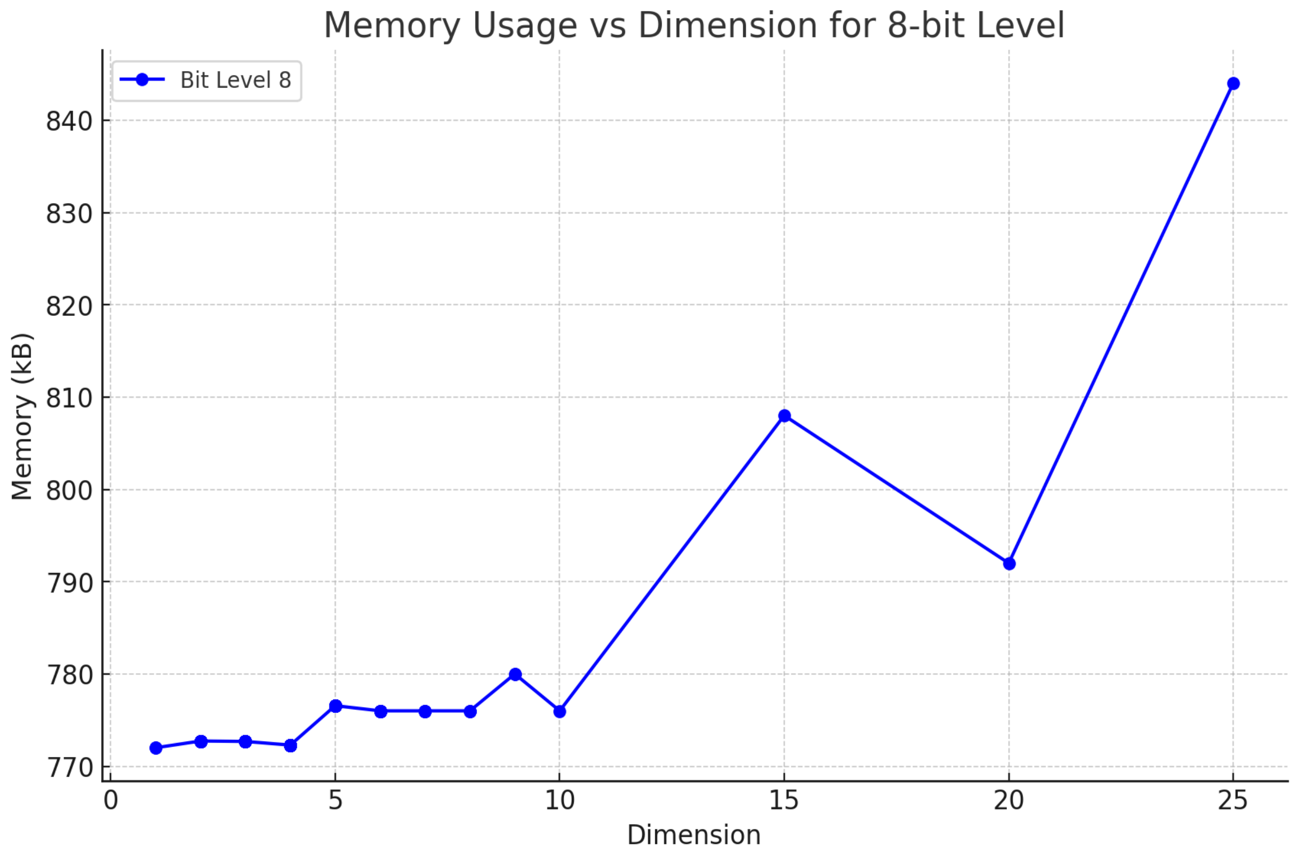
- n=25 -> 0.508 seconds

- n=30 -> 1.438 second

The Performance of the algorithm can be analyzed both in terms of memory and running time. Below the graphs shows the trade of between memory usage vs dimensions and running time vs dimensions.

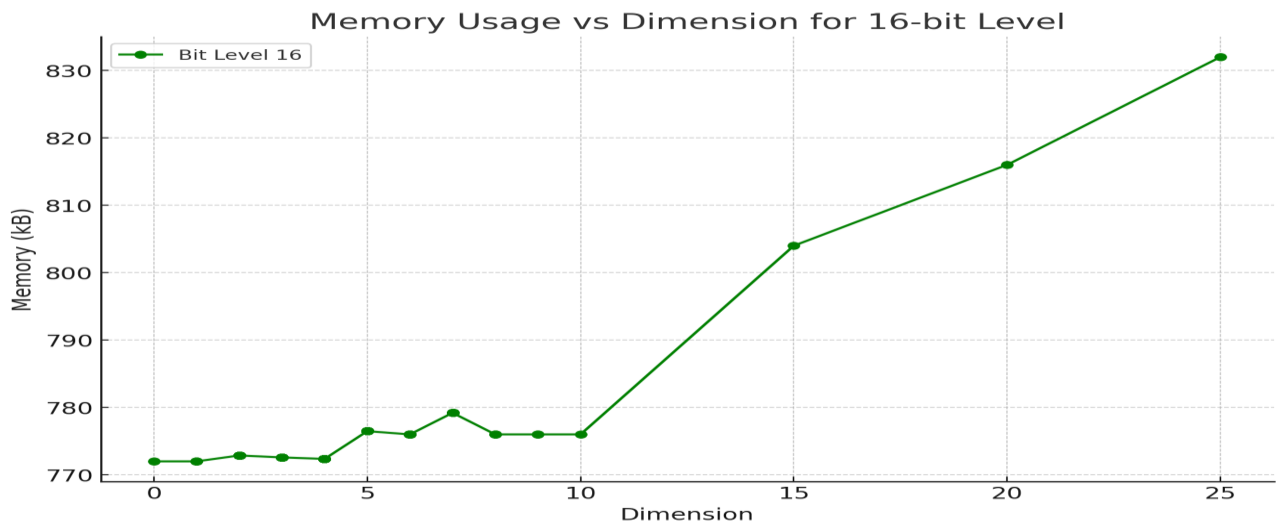
- **Memory usage analysis:**
  The graphs show three precision levels (8-bit, 16-bit, 32bit):

Graph 1: Memory Usage vs Dimension for 8-bit Level



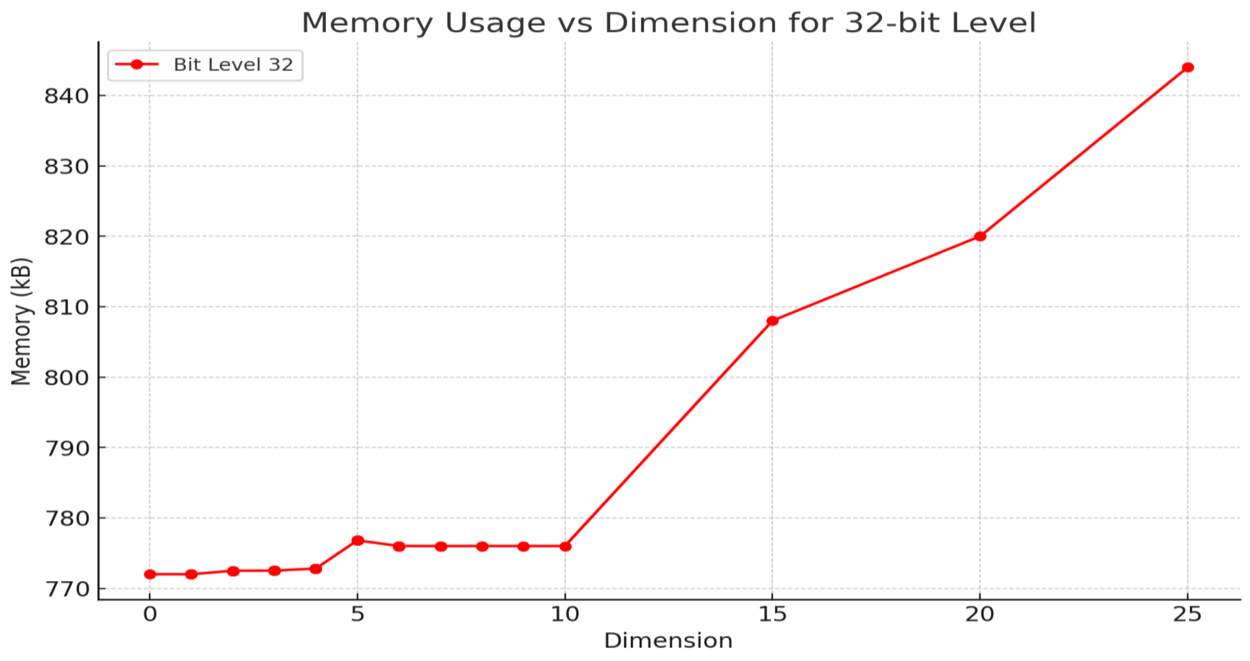
This graph shows a consistent memory usage across dimensions with a noticeable spike for large dimensions.

Graph 2: Memory Usage Dimension for 16-bit Level



Memory usage increases more linearly with dimension in the 16-bit level, suggesting a direct correlation between the two.

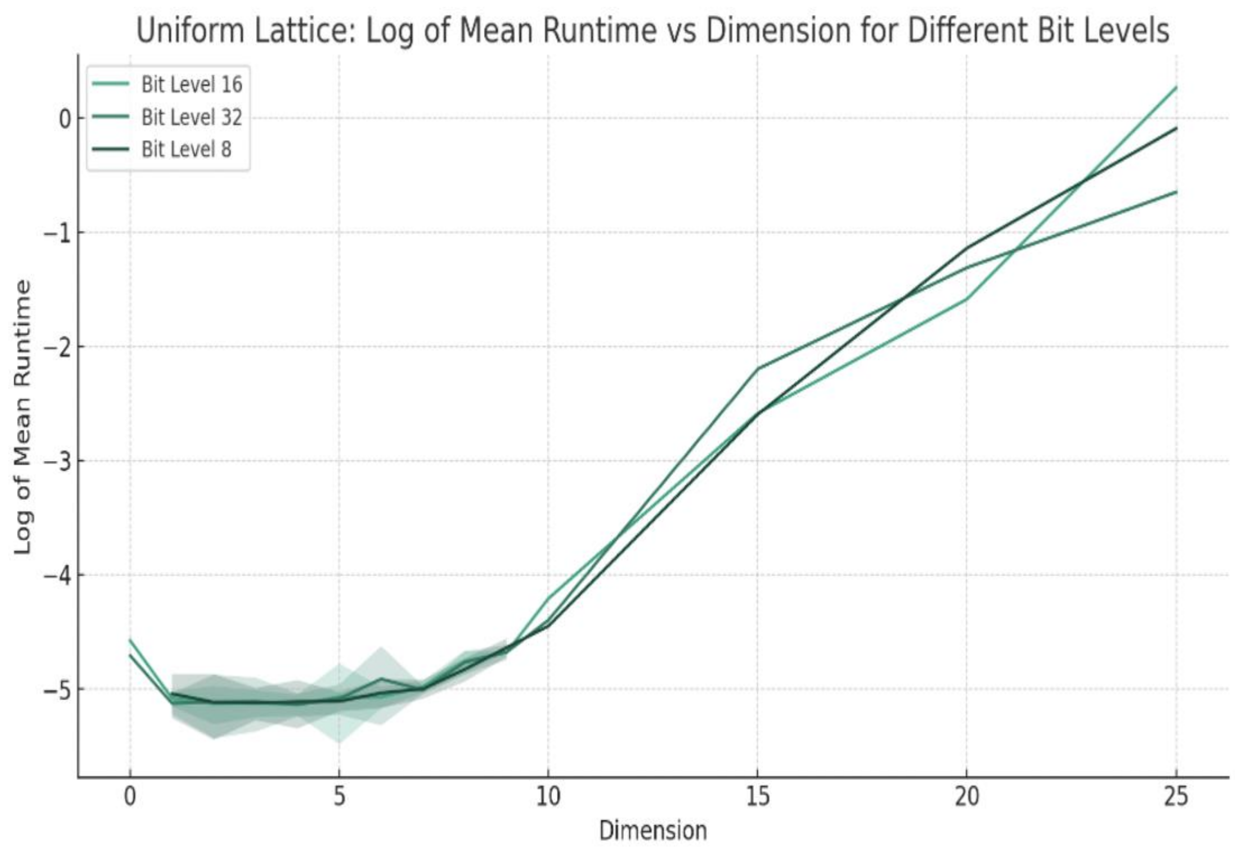
Graph 3: Memory Usage vs Dimension for 32-bit Level



This graph shows a significant increase as the dimension grows indicating that higher precision impact on the memory.

Runtime Performance Analysis:

Graph 4: Log of Mean Runtime vs Dimension for Different Bit Levels



The runtime performance, as shown in the log graph indicates that as dimensions increases the mean of runtime increases. This is consitent across all bit levels, with higher bit with higher bit levels showing slightly icreased runtime because computation of larger numbers.