International Journal of Information Science and Computer Mathematics

Available online at http://pphmj.com/journals/ijiscm.htm
Volume 7, Numbers 1-2, 2013, Pages 65-81

THE CONSTRUCTION OF GREEDY SVP LLL ALGORITHM

Saiful Khair, Sugi Guritman and Bib P. Silalahi

Department of Mathematics
Bogor Agricultural University
Dramaga Raya, No 41 Ciherang
Bogor, Indonesia

Abstract

LLL algorithm is an algorithm used to compute the approximation of the shortest nonzero vector in a basis of lattice. Terms of reduction size and the exchanging process are the important steps in the LLL algorithm. In 1994, Schnoor and Euchner modified this LLL algorithm which later was named LLL Deep Insertion algorithm, where the exchanging process in this algorithm scheme was comparing the projection in the orthogonal complement after done a certain vector reduction. This paper provides a new variant of LLL algorithm which is named Greedy SVP LLL algorithm, that is, purely comparing the-$\mathbf{b}_j$ length (norm) of lattice vector with the-$\mathbf{b}_i$ length of lattice vector, for $i = 1, 2, 3, ..., j-1$, along with the vector insertion process conducted greedily. Thereafter, the calculation of the number of operation and testing for all three algorithms are conducted experimentally.

1. Introduction

A lattice is a set of all integer linear combination of a set of linearly

Received: November 13, 2013; Accepted: December 30, 2013
2010 Mathematics Subject Classification: 68W40.
Keywords and phrases: algorithm, LLL, deep insertion, greedy SVP.

independent vectors in $\mathbb{R}^n$. The independent vectors are called a *basis* of lattice. Any lattice can be generated from many bases, and these bases have the same cardinality [1].

The most fundamental and renowned problem is the Shortest Vector Problem. Furthermore, the abbreviation SVP is often used in this paper. SVP is a tracking problem of the shortest nonzero vector in a lattice with equivalent basis. In two dimensions, SVP problem has resolved exactly by Gauss' algorithm. Research on the worst-case complexity of Gauss' algorithm was conducted by Lagarias [3]. He showed this algorithm is polynomial with respect to its input. The complexity of Gauss' algorithm was also investigated more deeply by Valley [7].

When the lattice dimension is higher than two, one has to defined precisely as the idea of basis reduction. In 1982, Lenstra et al. gave a reduction algorithm for lattice of arbitrary dimension. This algorithm is the result of generalization of Gauss' algorithm [1]. This algorithm is called *LLL algorithm*. Reduced basis solution obtained from LLL algorithm still be an approximation and has polynomial running time of arbitrary dimension which large enough. Then, Schnoor and Euchner discussed the modified LLL algorithm at exchange step for increasing the accuracy of LLL-reduced basis output and applying it to the subset sum problem [6].

The purpose of this paper is constructing the Greedy SVP LLL algorithm, which is a development idea of Deep Insertion algorithm, as well as a new variant of the LLL algorithm. Furthermore, calculating the number of involved arithmetic operation is conducted, and then experimentally compared between LLL algorithm, LLL Deep Insertion algorithm and LLL Greedy SVP algorithm.

2. Preliminary

Here are the basic concepts of the lattice.

Definition 2.1. Let $\mathcal{B} = \{\mathbf{b}_1, \mathbf{b}_2, ..., \mathbf{b}_n\}$ be a set of n linearly independent vectors in vector space $\mathbb{R}^m$. The lattice that generated by $\mathcal{B}$ is a set of

$\mathcal{L}(\mathcal{B}) = \left\{\sum_{j=1}^{n} x_j \mathbf{b}_j / x_j \in \mathbb{Z}\right\}$ which its elements consist of all integer linear combinations of $\mathcal{B}$. In this case, $\mathcal{B}$ is a basis for $\mathcal{L}(\mathcal{B})$.

Basis $\mathcal{B}$ for the lattice $\mathcal{L}(\mathcal{B})$ can be represented as matrix **B** sized $m \times n$ which its columns are the vector $\mathbf{b}_j$,

$$\mathbf{B} = (\mathbf{b}_1\ \mathbf{b}_2\ ...\ \mathbf{b}_n).$$

Then $\mathcal{L}(\mathcal{B})$ can be written as multiplication of matrix $\mathcal{L}(\mathbf{B}) = \{\mathbf{B}x / x \in \mathbb{Z}^n\}$. In this case, **B** is a matrix form of $\mathcal{B}$.

Definition 2.2. Let $\mathcal{B} = \{\mathbf{b}_1, \mathbf{b}_2, ..., \mathbf{b}_n\}$ be a set of n linearly independent vectors in $\mathbb{R}^m$. Then it can be constructed the subsequence of n mutually orthogonal vector of $\mathcal{B}^* = \{\mathbf{b}_1^*, \mathbf{b}_2^*, ..., \mathbf{b}_n^*\}$, where $\mathbf{b}_1^* = \mathbf{b}_1$, $\mathbf{b}_j^* = \mathbf{b}_j - \sum_{i=1}^{j-1} \mu_{j,i} \mathbf{b}_i^*$ for $j = 2, 3, ..., n$ and $\mu_{j,i} = \dfrac{\mathbf{b}_j \cdot \mathbf{b}_i^*}{\mathbf{b}_i^* \cdot \mathbf{b}_i^*}$.

Definition 2.3. For $j = 1, 2, ..., n$, projection function π_j of vector space $V = \langle \mathcal{B}^* \rangle = \langle \mathcal{B} \rangle$ to vector subspace $\langle \{\mathbf{b}_j^*, \mathbf{b}_{j+1}^*, ..., \mathbf{b}_n^*\} \rangle$ is defined as $\pi_j(\mathbf{v}) = \sum_{i=j}^{n} \left(\dfrac{\mathbf{v} \cdot \mathbf{b}_i^*}{\mathbf{b}_i^* \cdot \mathbf{b}_i^*} \right) \mathbf{b}_i^*$. If we take value of $\mathbf{v} = \mathbf{b}_k$, $k = 1, 2, ..., n$, then obtained

$$\pi_j(\mathbf{b}_k) = \sum_{i=j}^{n} \left(\frac{\mathbf{v} \cdot \mathbf{b}_i^*}{\mathbf{b}_i^* \cdot \mathbf{b}_i^*} \right) \mathbf{b}_i^* = \begin{cases} \mathbf{0}, & \text{for } k < j, \\ \mathbf{b}_k^*, & \text{for } k = j, \\ \mathbf{b}_k^* + \sum_{i=j}^{k-1} \mu_{ki} \mathbf{b}_i^*, & \text{for } k > j. \end{cases}$$

3. LLL Algorithm

The definition of reduced basis δ is as follows:

Definition 3.1. A basis $\mathcal{B} = [\mathbf{b}_1, \mathbf{b}_2, ..., \mathbf{b}_n]$ in $\mathbb{R}^m$ is called *LLL reduced* with parameter $\boldsymbol{\delta}$ if it satisfies

(1) $| \mu_{ji} | \leq \frac{1}{2}$, for every integer i, j with $1 \leq i < j < n$,

(2) $\delta \| \pi_j(\mathbf{b}_j) \|^2 \leq \| \pi_j(\mathbf{b}_{j+1}) \|^2$, for $j = 1, 2, ..., n-1$,

where δ is a reduced parameter of real numbers with $\frac{1}{4} < \delta < 1$.

The first requirement is the reduced basis δ must "nearly orthogonal" and in its computation case, this requirement can be reached out by using the Gram-Schmidt's orthogonalization. While the second requirement is called *exchange step*, or used to called as *Lovasz condition*, which can be rewritten as $(\delta - \mu_{j+1,\,j}^2) \| \mathbf{b}_j^* \|^2 \leq \| \mathbf{b}_{j+1}^* \|^2$. This inequality explained that Gram-Schmidt's vectors of LLL reduced basis must ordered decreasing with decreasing factor $\delta - \mu_{j+1,\,j}^2$. If there is a pair of vector $(\mathbf{b}_j^*, \mathbf{b}_{j+1}^*)$ does not follow the Lovasz condition, then the exchange between vectors will be conducted and the orthogonalization process will be redone.

Algorithm 3.2 (LLL Algorithm)

Input: $\mathcal{B} = [\mathbf{b}_1, \mathbf{b}_2, ..., \mathbf{b}_n]$ basis for $\mathcal{L}(\mathcal{B})$ and $\frac{1}{4} < \delta < 1$.

Output: $\mathcal{B} = [\mathbf{b}_1, \mathbf{b}_2, ..., \mathbf{b}_n]$ is LLL reduced basis for $\mathcal{L}(\mathcal{B})$ and $\mathcal{B}^* = [\mathbf{b}_1^*, \mathbf{b}_2^*, ..., \mathbf{b}_n^*]$ is the result of Gram-Schmidt's orthogonalization process of $\mathcal{B}$.

(1) $\mathbf{b}_1^* := \mathbf{b}_1$

(2) $j := 2$

(3) While $j \leq n$ do

(4) $\mathbf{b}_j^* := \mathbf{b}_j$

(5) For $i := j - 1$ down to 1 do

(6) $$\mu_{j,i} := \frac{\mathbf{b}_j \cdot \mathbf{b}_i^*}{\mathbf{b}_i^* \cdot \mathbf{b}_i^*}$$

(7) $\mathbf{b}_j^* := \mathbf{b}_j^* - \mu_{j,i}\mathbf{b}_i^*$

(8) $\mathbf{b}_j := \mathbf{b}_j - \lfloor \mu_{j,i} \rceil \mathbf{b}_i$

(9) EndFor

(10) If $(\delta - (\mu_{j,j-1})^2) \| \mathbf{b}_{j-1}^* \|^2 > \| \mathbf{b}_j^* \|^2$ then

(11) If $j = 2$ then

(12) Swap $\mathbf{b}_1$ and $\mathbf{b}_2$

(13) $\mathbf{b}_1^* := \mathbf{b}_2$

(14) Else $j > 2$ then

(15) Swap $\mathbf{b}_{j-1}$ dan $\mathbf{b}_j$

(16) $j := j - 1$

(17) EndIf

(18) Else

(19) $j := j + 1$

(20) EndIf

(21) EndWhile

4. Deep Insertion Algorithm

In the LLL algorithm, the test for exchange is well organized step by step ($\mathbf{b}_j$ with $\mathbf{b}_{j-1}$), then by using the deep insertion method, this test can be conducted directly into $\mathbf{b}_j$ with $\mathbf{b}_k$ for $k = 1, 2, ..., j - 1$. Suppose that at a certain computation phase, ordered basis lattice is obtained as follows:

$$\mathbf{b}_1, \mathbf{b}_2, ..., \mathbf{b}_{k-1}, \mathbf{b}_k, \mathbf{b}_{k+1}, ..., \mathbf{b}_{j-1}, \mathbf{b}_j, \mathbf{b}_{j+1}, ..., \mathbf{b}_n.$$

The orthogonalization procedure of Gram-Schmidt is defined as $\mathbf{b}_j = \mathbf{b}_j^* + \sum_{i=1}^{j-1} \mu_{j,i}\mathbf{b}_i^*$ for $j = 1, 2, ..., n$. Because of $\mathbf{b}_1^*, \mathbf{b}_2^*, ..., \mathbf{b}_j^*$ orthogonal, then obtained $\| \mathbf{b}_j \|^2 = \| \mathbf{b}_j^* \|^2 + \sum_{i=1}^{j-1} \mu_{j,i}^2 \| \mathbf{b}_i^* \|^2$. If $\mathbf{b}_j$ is inserted into $\mathbf{b}_k$, then the ordered basis lattice become

$$\mathbf{b}_1, \mathbf{b}_2, ..., \mathbf{b}_{k-1}, \mathbf{b}_j, \mathbf{b}_k, \mathbf{b}_{k+1}, ..., \mathbf{b}_{j-1}, \mathbf{b}_{j+1}, ..., \mathbf{b}_n.$$

With fixed vectors $\mathbf{b}_1^*, \mathbf{b}_2^*, ..., \mathbf{b}_{k-1}^*$, while the orthogonalization procedure of Gram-Schmidt for $\mathbf{b}_k$ is renewed as

$$\hat{\mathbf{b}}_k^* = \mathbf{b}_j - \sum_{i=1}^{k-1} \mu_{j,i}\mathbf{b}_i^* \Leftrightarrow \mathbf{b}_j = \hat{\mathbf{b}}_k^* + \sum_{i=1}^{k-1} \mu_{j,i}\mathbf{b}_i^*,$$

then

$$\| \mathbf{b}_j \|^2 = \| \hat{\mathbf{b}}_k^* \|^2 + \sum_{i=1}^{k-1} \mu_{j,i}^2 \| \mathbf{b}_i^* \|^2 \Leftrightarrow \| \hat{\mathbf{b}}_k^* \|^2$$

$$= \| \mathbf{b}_j \|^2 - \sum_{i=1}^{k-1} \mu_{j,i}^2 \| \mathbf{b}_i^* \|^2. \quad \text{(i)}$$

For $k = j - 1$, then $\| \hat{\mathbf{b}}_{j-1}^* \|^2 = \| \mathbf{b}_j \|^2 - \sum_{i=1}^{j-2} \mu_{j,i}^2 \| \mathbf{b}_i^* \|^2 \Leftrightarrow \| \hat{\mathbf{b}}_{j-1}^* \|^2 = \| \mathbf{b}_j^* \|^2 + \mu_{j,j-1} \| \mathbf{b}_{j-1}^* \|^2$. For $k = j - 1$, the exchange step of deep insertion method is equal to the exchange step of LLL algorithm, that is, if $\| \hat{\mathbf{b}}_{j-1}^* \|^2 < \delta \| \mathbf{b}_{j-1}^* \|^2$, then $\mathbf{b}_j$ is swap with $\mathbf{b}_{j-1}$. Generally, for any value of $k = 1, 2, 3, ..., j - 1$, then equation (i) obtained as follows:

$$\| \hat{\mathbf{b}}_1^* \|^2 = \| \mathbf{b}_j \|^2$$

$$\| \hat{\mathbf{b}}_2^* \|^2 = \| \mathbf{b}_j \|^2 - \mu_{j,1}^2 \| \mathbf{b}_1^* \|^2$$

$$\vdots$$

$$\| \hat{\mathbf{b}}_{j-1}^* \|^2 = \| \mathbf{b}_j \|^2 - \mu_{j,1}^2 \| \mathbf{b}_1^* \|^2 - \mu_{j,2}^2 \| \mathbf{b}_1^* \|^2 - \cdots - \mu_{j,j-2}^2 \| \mathbf{b}_{j-2}^* \|^2.$$

These equations can be used to state the value of k so that $\mathbf{b}_j$ can be inserted into $\mathbf{b}_k$. This can be happened if $\| \hat{\mathbf{b}}_k^* \|^2 < \delta \| \mathbf{b}_k^* \|^2$ and $\| \hat{\mathbf{b}}_{j-1}^* \|^2$ can be calculated recursively with explanation as follows. Define initial $C = \| \mathbf{b}_j \|$ and $k = 1$, then recursively calculate $C = C - \mu_{j,k}^2 \| \mathbf{b}_k^* \|^2$ and $k := k + 1$ and the process end when $C < \delta \| \mathbf{b}_k^* \|^2 \Leftrightarrow \| \hat{\mathbf{b}}_k^* \|^2 < \delta \| \mathbf{b}_k^* \|^2$ [2].

Algorithm 4.1 (LLL Deep Insertion Algorithm)

Input: $\mathcal{B} = [\mathbf{b}_1, \mathbf{b}_2, ..., \mathbf{b}_n]$ basis for $\mathcal{L}(\mathcal{B})$ and $\frac{1}{4} < \delta < 1$.

Output: $\mathcal{B} = [\mathbf{b}_1, \mathbf{b}_2, ..., \mathbf{b}_n]$ is a LLL reduced basis for $\mathcal{L}(\mathcal{B})$ and $\mathcal{B}^* = [\mathbf{b}_1^*, \mathbf{b}_2^*, ..., \mathbf{b}_n^*]$ is the result of Gram-Schmidt's orthogonalization process of $\mathcal{B}$.

(1) $\mathbf{b}_1^* := \mathbf{b}_1$

(2) $j := 2$

(3) While $j \le n$ do

(4) $\mathbf{b}_j^* := \mathbf{b}_j$

(5) For $i := j - 1$ down to 1 do

(6) $N_i := \mathbf{b}_i^* \cdot \mathbf{b}_i^*$

(7) $\mu_{j,i} := \dfrac{\mathbf{b}_j \cdot \mathbf{b}_i^*}{N_i}$

(8) $\mathbf{b}_j := \mathbf{b}_j - \lfloor \mu_{j,i} \rceil \mathbf{b}_i$

(9) $\mu_{j,i}^* := \dfrac{\mathbf{b}_j^* \cdot \mathbf{b}_i^*}{N_i}$

(10) $\mathbf{b}_j^* := \mathbf{b}_j^* - \mu_{j,i}^* \mathbf{b}_i^*$

(11) EndFor

(12) $C := \mathbf{b}_j \cdot \mathbf{b}_j$ (*Deep Insertion*)

(13) $k := 1$

(14) While $k < j$ do

(15) $h := \mathbf{b}_k^* \cdot \mathbf{b}_k^*$

(16) If $C < \delta h$ then

(17) If $k = 1$ then

(18) Insert $\mathbf{b}_j$ into 1st position

$$(\mathbf{b}_j, \mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_{j-1}, \mathbf{b}_{j+1}, ..., \mathbf{b}_n)$$

(19) $\mathbf{b}_1^* := \mathbf{b}_j$

(20) Else

(21) Insert $\mathbf{b}_j$ into kth position

$$\mathbf{b}_1, \mathbf{b}_2, ..., \mathbf{b}_{k-1}, \mathbf{b}_j, \mathbf{b}_{k+1}, ..., \mathbf{b}_{j-1}, \mathbf{b}_{j+1}, ..., \mathbf{b}_n$$

(22) $\mathbf{b}_k^* := \mathbf{b}_j$

(23) For $i := k - 1$ down to 1 do

(24) $N_i := \mathbf{b}_i^* \cdot \mathbf{b}_i^*$

(25) $\mu_{k,i}^* := \dfrac{\mathbf{b}_k^* \cdot \mathbf{b}_i^*}{N_i}$

(26) $\mathbf{b}_k^* := \mathbf{b}_k^* - \mu_{k,i}^* \mathbf{b}_i^*$

(27) EndFor

(28) EndIf

(29) Else

(30) $z := \mathbf{b}_j \cdot \mathbf{b}_k^*$

(31) $C := C - \dfrac{z^2}{h}$

(32) $k := k + 1$

(33) EndIf

(34) EndWhile

(35) $j := j + 1$

(36) EndWhile

5. Greedy SVP LLL Algorithm

The fundamental idea of greedy methods is as follows. If the smallest vector is in the first position, then the insertion will occur only in the second position or more; if two of the smallest vectors are ready in the first and second position, then the insertion will only occur in the third position or more, and so on. If the smallest vectors that are obtained by order are faster, then the algorithm will be done as soon as possible. With this fundamental idea, hoping these smallest vectors can be provided greedily. In this algorithm, the exchange step (insertion) is not based on the comparison of the projected vector in orthogonal complement $[\mathbf{b}_1, \mathbf{b}_2, ..., \mathbf{b}_{k-1}]^{\perp}$ after jth reduction (deep insertion method), but the insertion that conducted purely by comparing norm of lattice $\mathbf{b}_j$ with norm lattice $\mathbf{b}_i$ for $i = 1, 2, 3, ..., j-1$. Beside that, the insertion has done greedily.

Here are outline on how the algorithm works:

1. For $[\mathbf{b}_1]$, defined $\mathbf{b}_1^* = \mathbf{b}_1$, find the vector $\mathbf{b}_j$ of the reduction result $[\mathbf{b}_1]$ and $[\mathbf{b}_1^*]$ toward $[\mathbf{b}_2, \mathbf{b}_3, ..., \mathbf{b}_n]$ with the smallest norm. If $\| \mathbf{b}_j \| < \| \mathbf{b}_1 \|$, insert $\mathbf{b}_j, \mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_{j-1}, \mathbf{b}_{j+1}, ..., \mathbf{b}_n$, then we obtained the new $\mathbf{b}_1 = \mathbf{b}_j$ and the process is repeated again. But if $\| \mathbf{b}_1 \| \leq \| \mathbf{b}_j \|$, then insert $\mathbf{b}_1, \mathbf{b}_j, \mathbf{b}_2, \mathbf{b}_{j-1}, \mathbf{b}_{j+1}, ..., \mathbf{b}_n$ so that we obtained the new order of $[\mathbf{b}_1, \mathbf{b}_2]$ with the smallest size in the sequence. Then, compute $\mathbf{b}_2^*$ from input $\mathbf{b}_2$ and $\mathbf{b}_1^*$ so that we obtained the sequence $[\mathbf{b}_1^*, \mathbf{b}_2^*]$ and continued to the second step.

2. From $[\mathbf{b}_1, \mathbf{b}_2]$ and $[\mathbf{b}_1^*, \mathbf{b}_2^*]$, find the vector $\mathbf{b}_j$ as the reduction result of $[\mathbf{b}_1, \mathbf{b}_2]$ toward $[\mathbf{b}_3, \mathbf{b}_4, ..., \mathbf{b}_n]$ with the smallest norm. If $\| \mathbf{b}_j \| < \| \mathbf{b}_1 \|$, insert $\mathbf{b}_j, \mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_{j-1}, \mathbf{b}_{j+1}, ..., \mathbf{b}_n$ or if $\| \mathbf{b}_1 \| \leq \| \mathbf{b}_j \| < \| \mathbf{b}_2 \|$, insert $\mathbf{b}_1, \mathbf{b}_j, \mathbf{b}_2, \mathbf{b}_{j-1}, \mathbf{b}_{j+1}, ..., \mathbf{b}_n$ then back to the first step. But if $\| \mathbf{b}_2 \| \leq \| \mathbf{b}_j \|$ insert $\mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_j, \mathbf{b}_3, \mathbf{b}_{j-1}, \mathbf{b}_{j+1}, ..., \mathbf{b}_n$ so that we obtained the new order of $[\mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_3]$ with the smallest size in the sequence. Then compute $\mathbf{b}_3^*$ of input $\mathbf{b}_3$ and $[\mathbf{b}_1^*, \mathbf{b}_2^*]$ so that we obtained the sequence $[\mathbf{b}_1^*, \mathbf{b}_2^*, \mathbf{b}_3^*]$ and continued to the third step.

3. Generally, the kth of $[\mathbf{b}_1, \mathbf{b}_2, ..., \mathbf{b}_k]$ and $[\mathbf{b}_1^*, \mathbf{b}_2^*, ..., \mathbf{b}_k^*]$, find the vector $\mathbf{b}_j$ as the reduction result of $[\mathbf{b}_1, \mathbf{b}_2, ..., \mathbf{b}_k]$ toward $[\mathbf{b}_{k+1}, \mathbf{b}_{k+2}, ..., \mathbf{b}_n]$ with the smallest norm. Then insert $\mathbf{b}_j$ to $[\mathbf{b}_1, \mathbf{b}_2, ..., \mathbf{b}_n]$. If the insertion format $\mathbf{b}_j, \mathbf{b}_1, \mathbf{b}_2, ..., \mathbf{b}_k$ or $\mathbf{b}_1, \mathbf{b}_j, \mathbf{b}_2, ..., \mathbf{b}_k$ then back to the first step, and if the insertion format $\mathbf{b}_1, \mathbf{b}_2, ..., \mathbf{b}_{i-1}, \mathbf{b}_j, \mathbf{b}_i, ..., \mathbf{b}_k$ so that we obtained the new $[\mathbf{b}_1, \mathbf{b}_2, ..., \mathbf{b}_i]$, then from $\mathbf{b}_i$ and $[\mathbf{b}_1^*, \mathbf{b}_2^*, ..., \mathbf{b}_{i-1}^*]$ compute $\mathbf{b}_i^*$ to provide the new $[\mathbf{b}_1^*, \mathbf{b}_2^*, ..., \mathbf{b}_i^*]$, then back to the ith step. But if the insertion format $\mathbf{b}_1, \mathbf{b}_2, ..., \mathbf{b}_k, \mathbf{b}_j$, then we obtained the new order $[\mathbf{b}_1, \mathbf{b}_2, ..., \mathbf{b}_{k+1}]$ with the smallest size in the sequence. Then, compute $\mathbf{b}_{k+1}^*$ of input $\mathbf{b}_{k+1}$ and $[\mathbf{b}_1^*, \mathbf{b}_2^*, ..., \mathbf{b}_k^*]$ so that we obtained the sequence $[\mathbf{b}_1^*, \mathbf{b}_2^*, ..., \mathbf{b}_{k+1}^*]$ and continued to the $(k+1)$th step.

4. And so on, and the process terminated when $k = n$.

Algorithm 5.1 (Greedy SVP LLL Algorithm)

Input: $\mathcal{B} = [\mathbf{b}_1, \mathbf{b}_2, ..., \mathbf{b}_n]$ basis for $\mathcal{L}(\mathcal{B})$.

Output: $\mathcal{B} = [\mathbf{b}_1, \mathbf{b}_2, ..., \mathbf{b}_n]$ is the LLL reduced basis for $\mathcal{L}(\mathcal{B})$ and $\mathcal{B}^* = [\mathbf{b}_1^*, \mathbf{b}_2^*, ..., \mathbf{b}_n^*]$ is the result of Gram-Schmidt's orthogonalization process of $\mathcal{B}$.

(1) $\mathbf{b}_1^* := \mathbf{b}_1$

(2) $k := 1$

(3) While $k < n$ do:

(4) Variable initial for $[\mathbf{b}_1, ..., \mathbf{b}_k]$ and $[\mathbf{b}_{k+1}, ..., \mathbf{b}_n]$

(5) While $n - k$ do:

(6) $\mathbf{b}_y := \mathbf{b}_{k+1}$

(7) For $l = k$ down to 1 do

(8) $$\mu_{y,l} := \frac{\mathbf{b}_y \cdot \mathbf{b}_l^*}{\mathbf{b}_l^* \cdot \mathbf{b}_l^*}$$

(9) $\mathbf{b}_y := \mathbf{b}_y - \lfloor \mu_{y,l} \rceil \mathbf{b}_l$

(10) Endfor

(11) Compute $\| \mathbf{b}_y \|$

(12) $i := 1$

(13) For $j = 2, 3, ..., n - k$ do:

(14) Defined $\mathbf{b}_j$

(15) For $l = k$ down to 1 do

(16) $$\mu_{j,l} := \frac{\mathbf{b}_j \cdot \mathbf{b}_l^*}{\mathbf{b}_l^* \cdot \mathbf{b}_l^*}$$

(17) $\mathbf{b}_j := \mathbf{b}_j - \lfloor \mu_{j,l} \rceil \mathbf{b}_l$

(18) Endfor

(19) Compute $\| \mathbf{b}_j \|$

(20) If $\| \mathbf{b}_j \| < \| \mathbf{b}_y \|$ then

(21) $\mathbf{b}_y := \mathbf{b}_j$

(22) $\| \mathbf{b}_y \| := \| \mathbf{b}_j \|$

(23) $i := j$

(24) Endfor

(25) Defined $[\mathbf{b}_{k+2}, ..., \mathbf{b}_n]$

(26) $m := n - k - 1$

(27) $b := k + 1$

(28) For $z = 1$ to k do

(29) Compute $\| \mathbf{b}_z \|$

(30) If $\| \mathbf{b}_y \| < \| \mathbf{b}_z \|$ then

(31) $b := z$ (the position of vector $\mathbf{b}_y$ swap with the position of vector $\mathbf{b}_z$)

(32) Break (Stop loop)

(33) Endif

(34) Endfor

(35) If position $\mathbf{b}_y \leq \mathbf{b}_k$ then

(36) If $b = 1$ then

(37) Defined $k := 1$

(38) $\mathbf{b}_y, \mathbf{b}_1, \mathbf{b}_2, ..., \mathbf{b}_k$

(39) $\mathbf{b}_y := \mathbf{b}_1^*$

(40) Else

(41) $\mathbf{b}_1, ..., \mathbf{b}_{z-1}, \mathbf{b}_y, \mathbf{b}_{z+1}, ..., \mathbf{b}_k$

(42) Defined $\mathbf{b}_y$

(43) For $l = k$ downto 1 do

(44) $$\mu_{y,l} := \frac{\mathbf{b}_y \cdot \mathbf{b}_l^*}{\mathbf{b}_l^* \cdot \mathbf{b}_l^*}$$

(45) $$\mathbf{b}_y^* := \mathbf{b}_y^* - \mu_{y,l}\mathbf{b}_l^*$$

(46) Endfor

(47) Updated $\mathbf{b}_y^*, \mathbf{b}_{z+1}^*, ..., \mathbf{b}_k^*$

(48) $k := b$

(49) Endif

(50) Break (Stop loop)

(51) Endif

(52) Updated $[\mathbf{b}_1, \mathbf{b}_2, ..., \mathbf{b}_k, \mathbf{b}_y]$

(53) Defined $\mathbf{b}_y$

(54) For $l = k$ downto 1 do

(55) $$\mu_{y,l} := \frac{\mathbf{b}_y \cdot \mathbf{b}_l^*}{\mathbf{b}_l^* \cdot \mathbf{b}_l^*}$$

(56) $$\mathbf{b}_y^* := \mathbf{b}_y^* - \mu_{y,l}\mathbf{b}_l^*$$

(57) Endfor

(58) Updated $\mathbf{b}_1^*, \mathbf{b}_2^*, ..., \mathbf{b}_y^*$

(59) Updated $\mathbf{b}_1^*, \mathbf{b}_2^*, ..., \mathbf{b}_n^*$

(60) $k := k + 1$

(61) End While

(62) Updated $[\mathbf{b}_1, ..., \mathbf{b}_k, \mathbf{b}_{k+1}, ..., \mathbf{b}_n]$

(63) EndWhile

6. Speed Analysis and Implementation

Calculating the number of operation and complexity in the constructed Greedy SVP LLL algorithm are explained here. The algorithm begins

with the first vector initial as orthogonal vector, continued by assignment operation in the value $k := 1$. Then, initialization is done onto two variables to divide the column vector that is in the matrix of k value assignment. Initialization process in this step (4) is intend to compare vector one by one that is in the 2 variable. Entering into loop "while" that will repeated $n - k$ times, with value of n is the inputted matrix dimension.

Furthermore, the algorithm will compute the number of involving in the size reduction. The number of existing operations in Algorithm 5.1((6)-(10)) is as follows:

(1) An assignment operation as initial statement for yth vector that wanted to be reduced.

(2) There are block of statement "for" which are repeated as many as k-times.

(a) There are 2 assignment operations.

(b) There are 3 vector multiplication, 1 subtraction, 1 division, and 1 rounding operation to the closest integer.

The complexity in the block of reduction size is $O(n)$. After this block, the value of norm is calculated from reduced vector that given in a certain variable, then initialized by a certain variable i. The complexity for this step is $O(n)$.

In the steps (13)-(24), statement block initially by looping for reduction size for $k + 2$ th vector up to nth vector. The number of operations for this block as follows:

(1) Block for reduction size that using the same steps as steps (6) to (10) with the same complexity, that is $O(n)$.

(2) Compute every norm of reduced vectors for $k + 2$ th vector up to nth vector. Then, there is one branching in this block, where there is inequality for comparing the norm of reduced vector in step (6), to get the shortest one. In this block, there are 3 initialization, each of exchange position of vector with the shortest norm.

The complexity of this block is $O(n^2)$.

In the steps (25)-(27), assignment operation for $k+2$ th vector up to *n*th vector, and variables *m* and *b* stating the vector position. The complexity in this step is $O(1)$.

Furthermore, the steps (28)-(34) are looping to calculate the norm of vector position 1 to the *k*th vector and there is statement "if" where the smallest vector of the reduction result in the step (13) is compared by its norm. If this condition satisfies, then the vector position will be exchanged to the *k*th vector position. The complexity of this step is $O(n)$.

In the steps (35)-(52), there is branching block which allows to insert vector with the smallest norm to take the first position, or vector position that inserted between the first vector and the *k*th vector. If this condition satisfies, then the sequence that contains the smallest vector can be calculated its orthogonal vector by using the Gram-Schmidt's orthogonalization. This vector is passing through the long enough path way after passing the step (35) then turn we will go to the step (43) and end with the last step (50). The complexity in this step is $O(n^2)$.

Furthermore, in the steps (53)-(58), initialed by vector initialization that is not included in the branching condition, to calculate its orthogonal vectors. The details of the number operation in the statement block this "for" are:

(1) 2 initialed operation.

(2) 1 division, 3 multiplication vector, and 1 subtraction operation.

The complexity of this block is $O(n)$. The last step is adding the index *k* then back to the step (3) and combining the latest result of the vector that has been reduced and exchanged with its orthogonal vector.

In addition to counting the number of operation and complexity on the parts of algorithm, and the test towards LLL algorithm, deep insertion algorithm, and Greedy SVP LLL algorithm also conducted. The testing is done by inputting the integer matrix size $n \times n$ for $n = 10, 20, ..., 80$ with

$\delta = \frac{3}{4}$. The output are the integer matrix size $n \times n$ as the result of reduced LLL and the result of matrix Gram-Schimdt's orthogonalization. To obtain the value of running time, run those algorithms for each matrix size as many as 5 times and calculate the average of it. Here is the result:

Table 1. Matrix size $n \times n$ versus *running time* (sec) with $\delta = 3/4$

Algorithm	Matrix size							
	10×10	20×20	30×30	40×40	50×50	60×60	70×70	80×80
LLL	0.059	1.207	8.234	13.104	33.712	83.034	93.544	388.099
Deep Insertion	0.072	1.763	13.625	64.659	136.485	401.216	651.058	2126.497
Greedy SVP LLL	0.044	1.061	7.132	7.226	17.634	34.617	83.408	139.385

It can be presented in the graphic as follows:

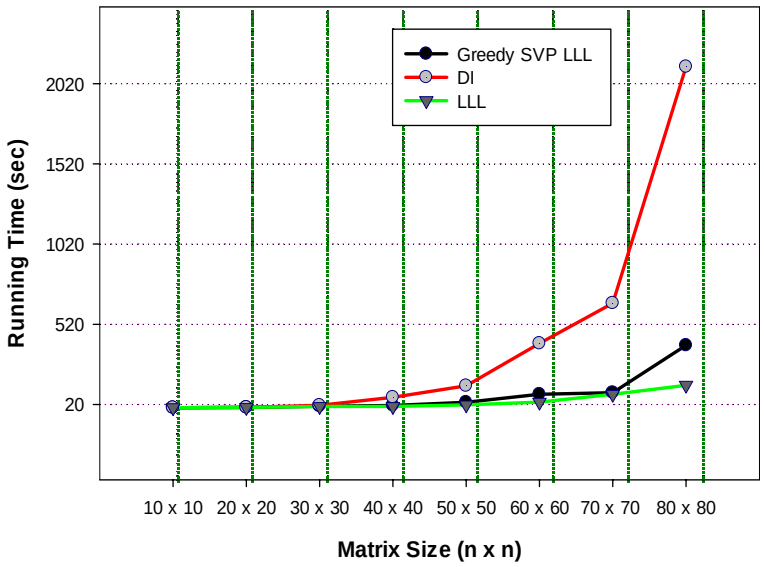

Figure 1. The comparison of *running time* (sec) versus matrix size $n \times n$.

7. Conclusion

With increasing in matrix size, the running time of the three algorithms has increased. The result of comparing experimentally shows that by using $\delta = \frac{3}{4}$ for the LLL algorithm and the deep insertion LLL algorithm, and the Greedy SVP LLL algorithm which is a new variant made by using no parameter of δ, outperform of the other of two algorithm in terms of speed with the same output.

References

[1] Ali Akhavi, The optimal LLL algorithm is still polynomial in fixed dimension, Theoret. Comput. Sci. 297 (2003), 3-23.

[2] M. R. Bremner, Lattice Basis Reduction: An Introduction to the LLL Algorithm and its Application, CRC Press, New York, 2012.

[3] J. C. Lagarias, Worst-case complexity bounds for algorithms in the integral quadratic forms, J. Algorithms 1 (1980), 142-186.

[4] A. K. Lenstra, H. W. Lenstra and L. Lovasz, Factoring polynomials with rational coefficients, Math. Ann. 261 (1982), 515-534.

[5] D. Micciancio and P. Voulgaris, Faster Exponential Time Algorithms for Shortest Vector Problem, SIAM, 2011, pp. 1468-1480.

[6] C. P. Schnoor and M. Euchner, Lattice basis reduction: improved practical algorithms and solving subset sum problems, Math. Program. 66 (1994), 181-199.

[7] B. Valley, Gauss algorithm revisited, J. Algorithms 12 (1991), 556-572.