CSE 206A: Lattice Algorithms and Applications Spring 2014

Basic Algorithms

Instructor: *Daniele Micciancio* UCSD CSE

We have already seen an algorithm to compute the Gram-Schmidt orthogonalization of a lattice basis. We continue our study of lattice algorithms by examining two other important important algorithms:

- Hemite normal form computation, which is useful to solve many set theoretic problems, like like testing membership in a lattice, inclusion or equality between two lattices, computing the union and intersection of lattices, etc.
- Various types of enumeration algorithms, based on standard algorithmic techniques like the greedy method, or branch and bound. These are useful to solve more geometric problems, like finding short lattice vectors, or lattice vectors close to a given target.

1 Hermite Normal Form

We have already described a method to compute the Hemite normal form of a nonsingular square matrix. However, the method has two drawbacks:

- The method cannot be applied to lattices that are not full rank, or linearly dependent sets of lattice vectors,
- It may not run in polynomial time because of number size explosion.

Here we give an efficient algorithm to compute the Hermite normal form of arbitrary integer matrices, and then use it to solve various lattice problems. First we extent our previous definition of Hermite Normal Form (HNF) from square to arbitrary matrices.[1]

Definition 1 *A non-singular matrix* $\mathbf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n] \in \mathbb{R}^{m \times n}$ *is in Hermite normal form (*HNF*) iff*

- *There exists* $1 \leq i_1 < \ldots < i_h \leq m$ *such that* $b_{i,j} \neq 0 \Rightarrow (j < h) \wedge (i \geq i_j)$ *(strictly decreasing column height).*
- *For all* $k > j$, $0 \leq b_{i_j,k} < b_{i_j,j}$, *i.e., all elements at rows* i_j *are reduced modulo* $b_{i_j,j}$.

The index h is the number of non-zero columns in the matrix, and index i_j is the row of the top non-zero element of column j. Because these terms are strictly increasing, each column contains rows that none of the later columns have. Thus, the non-zero columns of a matrix

[1] The definition given here corresponds to lower triangular HNF. The upper triangular HNF can be easily obtained by reversing the order of the vectors and their coordinates.

in HNF (as well as the rows with indices i_j) are guaranteed to be linearly independent. We also know (but will not show here) that HNF is unique, i.e., if two matrices $\mathbf{B}$ and $\mathbf{B}'$ are in HNF and they generate the same lattice ($\mathcal{L}(\mathbf{B}) = \mathcal{L}(\mathbf{B}')$), then $\mathbf{B} = \mathbf{B}'$ (except at most for the number of zero-columns at the end). We say that $\mathbf{H}$ is the HNF of $\mathbf{B}$ if $\mathcal{L}(\mathbf{H}) = \mathcal{L}(\mathbf{B})$, $\mathbf{H}$ is in HNF, and $\mathbf{H}$ does not contain zero columns.

It is not hard to come up with an algorith that computes the HNF of a matrix performing only a polynomial number of operations. However, a naive solution to this problem may result in superpolynomial running time because the numbers in the intermediate computations can easily get very large. In order to achieve polynomial running time, some care is required.

Notice that the problem of computing the HNF of a rational matrix $\mathbf{B} \in \mathbb{Q}^{m\times n}$ easily reduces to the problem of computing the HNF of an integer matrix as follows:

1. let D be the least common multiple of all denominators occurring in $\mathbf{B}$,
2. Compute the HNF $\mathbf{H}$ of integer matrix $D \cdot \mathbf{B} \in \mathbb{Z}^{m\times n}$
3. Output $D^{-1} \cdot \mathbf{H}$

It is easy to verify that that $\mathbf{H}$ is in HNF if and only if $D\cdot\mathbf{B}$ is. Moreover, if $\mathbf{H}$ and $D\cdot\mathbf{B}$, generate the same lattice, then $D^{-1} \cdot \mathbf{H}$ and $\mathbf{B}$ also generate the same lattice. Therefore, $D^{-1} \cdot \mathbf{H}$ is the HNF of $\mathbf{B}$. This reduction is polynomial time because $\log_2 D$ is at most as big as the bitsize of the input matrix. In the rest of this section we show how to compute the HNF of an integer matrix.

So, we may assume that the input matrix has integer entries. We first give an algorithm to compute the HNF of matrices with full row rank, and then show how to adapt it to arbitrary matrices.

1.1 Matrices with full row rank

We give an algorithm that can find the HNF of any matrix $\mathbf{B} \in \mathbb{Z}^{m\times n}$ which has full row rank. We know that in this case the HNF of $\mathbf{B}$ is a square non-singular $m \times m$ matrix $\mathbf{H}$. The idea is to first find the HNF basis $\mathbf{H}$ of a sublattice of $\mathcal{L}(\mathbf{B})$, and then update $\mathbf{H}$ by including the columns of $\mathbf{B}$ one by one. Let's assume for now that we have a polynomial time procedure AddColumn that on input a square non-singular HNF matrix $\mathbf{H} \in \mathbb{Z}^{m\times m}$ and a vector $\mathbf{b}$outputs the HNF of matrix $[\mathbf{H}|\mathbf{b}]$. The HNF of $\mathbf{B}$ can be computed as follows:

1. Apply the Gram-Schmidt algorithm to the *columns* of $\mathbf{B}$ to find m linearly independent columns. Let $\mathbf{B}'$ be the $m \times m$ matrix given by these columns.
2. Compute $d = \det(\mathbf{B}')$, using the Gram-Schmidt algorithm or any other polynomial time procedure. Let $\mathbf{H}_0 = d \cdot \mathbf{I}$ be the diagonal matrix with d on the diagonal.
3. For $i = 1, \ldots, n$ let $\mathbf{H}_i$ the result of applying AddColumn to input $\mathbf{H}_{i-1}$ and $\mathbf{b}_i$.
4. Output $\mathbf{H}_n$.

The correctness of the algorithm is based on the invariant that for all i,$\mathbf{H}_i$ is the HNF of the lattice $\mathcal{L}([d \cdot \mathbf{I}|\mathbf{b}_1, \ldots, \mathbf{b}_i])$. The invariant is clearly satisfied for $i = 0$. Moreover, it is preserved at every iteration by definition of AddColumn. So, upon termination, the algorithm outputs the HNF of $\mathcal{L}([d \cdot \mathbf{I}|\mathbf{B}])$. Finally, since $d \cdot \mathbb{Z}^m \subseteq \mathcal{L}(\mathbf{B}') \subseteq \mathcal{L}(\mathbf{B})$, we have $\mathcal{L}([d \cdot \mathbf{I}|\mathbf{B}]) = \mathcal{L}(\mathbf{B})$ and the algorithm outputs the HNF of $\mathbf{B}$.

Notice that during the entire process, all the entries of $\mathbf{H_i}$ stay bounded by d. In particular, all the numbers are polynomial in the original input $\mathbf{B}$. So, if AddColumn is polynomial time, then the entire HNF algorithm is polynomial time. It remains to give an algorithm for the AddColumn procedure. On input a square non-singular HNF matrix $\mathbf{H} \in \mathbb{Z}^{m \times m}$ and a vector $\mathbf{b} \in \mathbb{Z}^m$, AddColumn proceeds as follows. If $m = 0$, then there is nothing to do, and we can immediately terminate with output $\mathbf{H}$. Otherwise, let $\mathbf{H}= \begin{bmatrix} a & \mathbf{0}^\top \\ \mathbf{h} & \mathbf{H}' \end{bmatrix}$ and $\mathbf{b} = \begin{bmatrix} b \\ \mathbf{b}' \end{bmatrix}$ and proceed as follows:

1. Compute $g = \gcd(a, b)$ and integers x, y such that $xa + yb = g$ using the extended gcd algorithm.
2. Apply the unimodular transformation $\mathbf{U} = \begin{bmatrix} x & (-b/g) \\ y & (a/g) \end{bmatrix}$ to the first column of $\mathbf{H}$ and $\mathbf{b}$ to obtain
$$\begin{bmatrix} a & b \\ \mathbf{h} & \mathbf{b}' \end{bmatrix} \mathbf{U} = \begin{bmatrix} g & 0 \\ \mathbf{h}' & \mathbf{b}'' \end{bmatrix}.$$
3. Add an appropriate vector from $\mathcal{L}(\mathbf{H}')$ to $\mathbf{b}''$ so to reduce its entries modulo the diagonal elements of $\mathbf{H}'$.
4. Recursively invoke AddColumn on input $\mathbf{H}'$ and $\mathbf{b}''$ to obtain a matrix $\mathbf{H}''$
5. Add an appropriate vector from $\mathcal{L}(\mathbf{H}'')$ to $\mathbf{h}'$ so to reduce its entries modulo the diagonal elements of $\mathbf{H}''$
6. Output $\begin{bmatrix} g & \mathbf{0}^\top \\ \mathbf{h}' & \mathbf{H}'' \end{bmatrix}$

1.2 General case

We would like to reduce the general case to the full-dimensional case. We begin by using a projection operation Π to select $\mathbf{B}'$, a submatrix of $\mathbf{B}$ consisting only of linearly independent rows of $\mathbf{B}$. Then we use the algorithm for the full-dimensional case. Finally, we use the inverse of the projection operation to get our final result.

1. Run the Gram-Schmidt orthogonalization process on the *rows* $\mathbf{r}_1, \ldots \mathbf{r}_m$ of $\mathbf{B}$, and let $K = \{k_1, \ldots, k_l\}$ $(k_1 < \ldots < k_l)$ be the set of indices such that $\mathbf{r}^*_{k_i} \neq \mathbf{0}$. Define the projection operation $\Pi_K : \mathbb{R}^m \to \mathbb{R}^l$ by $[\Pi_K(\mathbf{x})]_i = x_{k_i}$. Notice that the rows $\mathbf{r}_k$ $(k \in K)$ are linearly independent and any other row can be expressed as a linear combination of the previous rows $\mathbf{r}_j$ $(\{j \in K : j < i\})$. Therefore Π_K is one-to-one when restricted

to $\mathcal{L}(\mathbf{B})$, and its inverse can be easily computed using the Gram-Schmidt coefficients $\mu_{i,j}$.

2. Define a new matrix $\mathbf{B}' = \Pi_K(\mathbf{B})$, which is full-rank, and run the algorithm given in the previous section to find the HNF$\mathbf{B}''$ of $\mathbf{B}'$.

3. Apply the inverse projection function, Π_K^{-1}, to the HNFdetermined in the previous step ($\mathbf{B}''$), to give matrix $\mathbf{H}$. It is easy to see that $\mathcal{L}(\mathbf{H})\mathcal{L}(\mathbf{B})$ and $\mathbf{H}$ is in HNF. Therefore $\mathbf{H}$ is the HNF of $\mathbf{B}$.

This gives us an algorithm for determining the HNF that runs in time polynomial in n, m and $log(d)$. To complete the proof we need to show that $\log(d)$ is polynomial in the bit-size of the original matrix. Since d is the determinant of a submatrix of B, it is enough to show that for any square matrix $A \in \mathbb{Z}^{n\times n}$, size(det($A$)) is polynomial in size(A). Using the Hadamerd inequality $\text{vol}(\mathcal{P}(A)) \leq \Pi\|a_i\|$, we can write

$$d = |\det(A)| \leq \Pi\|\mathbf{a_i}\|$$

If all a_{ij} have bit-size at most α (i.e., $\lg|b_{ij}| \leq \alpha$), we get $\lg d \leq n(\alpha + \lg\sqrt{m})$, proving that the size of the determinant is polynomial in the size of the matrix.

1.3 Applications

We use the HNF algorithm and the dual lattice to efficiently solve various basic problems on lattices.

Basis problem Given a set of rational vectors $\mathbf{B}$, we want to compute a basis for the lattice $\mathcal{L}(\mathbf{B})$.

This problem is immediately solved (in polynomial time) by computing HNF($\mathbf{B}$).

Equivalence problem Given two bases $\mathbf{B}$ and $\mathbf{B}'$, we want to determine if they define the same lattice $\mathcal{L}(\mathbf{B}') = \mathcal{L}(\mathbf{B})$.

This problem can be solved in polynomial time by computing $\mathbf{H} = \text{HNF}(\mathbf{B})$ and $\mathbf{H}' = \text{HNF}(\mathbf{B}')$, and checking if $\mathbf{H} = \mathbf{H}'$.

Union of lattices Given two bases $\mathbf{B}$ and $\mathbf{B}'$, we want to determine a basis for the smallest lattice containing both $\mathcal{L}(\mathbf{B})$ and $\mathcal{L}(\mathbf{B}')$.

It is easy to see that this lattice is generated by $[\mathbf{B} \mid \mathbf{B}']$, so a basis for the lattice can be easily computed as HNF($[\mathbf{B} \mid \mathbf{B}']$).

Containment problem Given two bases $\mathbf{B}$ and $\mathbf{B}'$, we want to determine if $\mathcal{L}(\mathbf{B}')$ is a sublattice of $\mathcal{L}(\mathbf{B})$, i.e., $\mathcal{L}(\mathbf{B}') \subseteq \mathcal{L}(\mathbf{B})$.

This problem is easily reduced to the union and equivalence problems: $\mathcal{L}(\mathbf{B}') \subseteq \mathcal{L}(\mathbf{B})$ if and only if $\mathcal{L}([\mathbf{B} \mid \mathbf{B}']) = \mathcal{L}(\mathbf{B})$. So, in order to check the inclusion we only need to compute $\mathrm{HNF}([\mathbf{B} \mid \mathbf{B}'])$ and $\mathrm{HNF}(\mathbf{B})$ and check for equality of the HNF bases.

Membership problem Given a lattice $\mathbf{B}$ and a vector $\mathbf{v}$, we want to determine if $\mathbf{v} \in \mathcal{L}(\mathbf{B})$.

This immediately reduces to the containment problem, by checking if $\mathcal{L}([\mathbf{v}]) \subseteq \mathcal{L}(\mathbf{B})$. If we need to check membership for many vectors $\mathbf{v}_1, \ldots, \mathbf{v}_n$, then it is convenient to first compute $\mathbf{H} = \mathrm{HNF}(\mathbf{B})$, and then for every i check if $\mathbf{H} = \mathrm{HNF}([\mathbf{H} \mid \mathbf{v}_i])$. Notice that $\mathrm{HNF}([\mathbf{H} \mid \mathbf{v}_i])$ can be computed much faster than a HNF matrix computation because most of the matrix is already in Hermite Normal Form.

Solving linear system Let $\mathbf{Ax} = \mathbf{b}$ be a system of linear equations. We want to find a solution $\mathbf{x}$, or tell if no solution exists. (We know this can be done in $O(n^3)$ arithmetic operations. The problem is to show that the numbers stay small.)

We can easily tell if the system admit solution by running Gram-Schmidt on $[\mathbf{A} \mid \mathbf{b}]$, and checking that the last column equals $\mathbf{b}^* = \mathbf{0}$. Also, we can restrict our attention to solutions that only use variables for which $\mathbf{a}^* \neq \mathbf{0}$. So, assume that the columns of $\mathbf{A}$ are linearly independent, and $\mathbf{Ax} = \mathbf{b}$ for some $\mathbf{x}$. We can also eliminate redundant equations by running Gram-Schmidt on the rows of $[\mathbf{A} \mid \mathbf{b}]$ and throwing away equations for which the corresponding orthogonalized vector is $\mathbf{0}$.

So far we have reduced the problem of solving an arbitrary system of linear equations, to solving a system $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{A}$ is a non-singular square matrix. Then, the system can be easily solved by computing the HNF (or, equivalently Gram-Schmidt orthogonalization) of $[\mathbf{A} \mid \mathbf{b}]^T$ to yield $[\mathbf{C} \mid \mathbf{d}]^T$ where $\mathbf{C}$ is a triangular matrix, and then solve the equivalent triangular system $\mathbf{Cx} = \mathbf{d}$ by back substitution. It is easy to see that in this case all the numbers involved are guaranteed not to get too big, and the system can be solved in polynomial time. (Note: this is certainly not the fastest way to solve a system of linear equations. Much faster methods are known.)

As a special case, this shows that the inverse of a non-singular square matrix $\mathbf{A}$ can be computed in polynomial time by solving the equations $\mathbf{Ax}_i = \mathbf{e}_i$. The inverse matrix is given by $[\mathbf{x}_1, \ldots, \mathbf{x}_n]$.

Dual Lattice Given a lattice basis $\mathbf{B}$, compute the dual basis $\mathbf{D}$, i.e., a basis $\mathbf{D}$ such that $\mathbf{B}^\top \mathbf{D} = \mathbf{I}$ and $\mathrm{span}(\mathbf{B}) = \mathrm{span}(\mathbf{D})$.

This problem is easily solved by computing $\mathbf{D}$ as $\mathbf{D} = \mathbf{B}(\mathbf{B}^T\mathbf{B})^{-1}$. Notice that this computation involves three matrix multiplications, and one matrix inversion.

Intersection of lattices Given two bases $\mathbf{B}$ and $\mathbf{B}'$, we want to determine a basis for the intersection $\mathcal{L}(\mathbf{B}) \cap \mathcal{L}(\mathbf{B}')$.

It is easy to show that if $\mathcal{L}(\mathbf{D})$ and $\mathcal{L}(\mathbf{D}')$ are the dual lattices of $\mathcal{L}(\mathbf{B})$ and $\mathcal{L}(\mathbf{B}')$, then the dual of $\mathcal{L}(\mathbf{B}) \cap \mathcal{L}(\mathbf{B}')$ is $\mathcal{L}([\mathbf{D} \mid \mathbf{D}'])$. So, a basis for the intersection is obtained by first computing $\mathbf{D}, \mathbf{D}'$ and $\mathbf{H} = \mathrm{HNF}([\mathbf{D} \mid \mathbf{D}'])$, and then computing the dual of $\mathbf{H}$.

Cyclic lattice Let $r(\mathbf{x})$ be the cyclic rotation of vector $\mathbf{x}$, i.e., $r(x_1, \ldots, x_n) = (x_n, x_1, x_2, \ldots, x_{n-1})$. Given a set of vectors $\mathbf{B}$, find the cyclic lattice generated by $\mathbf{B}$, i.e., the smallest cyclic lattice containing $\mathbf{B}$.

This problem is easily solved considering the vectors $r^i(\mathbf{b}_j)$ for all $i = 0, \ldots, n-1$ and $\mathbf{b}_j \in \mathbf{B}$, and computing the lattice generated by these vectors.

A similar problem is that of deciding if a given lattice $\mathcal{L}(\mathbf{B})$ is cyclic, i.e., if $r(\mathcal{L}(\mathbf{B})) \subseteq \mathcal{L}(\mathbf{B})$ (in which case, we also have $r(\mathcal{L}(\mathbf{B})) = \mathcal{L}(\mathbf{B})$. The cyclic lattice decision problem is easily solved computing $\mathrm{HNF}(\mathbf{B})$ and $\mathrm{HNF}(r(\mathbf{B}))$ and cheking these two matrices for equality.

2 Enumeration Algorithms

Consider the Closest Vector Problem (CVP): Given a lattice basis $\mathbf{B} \in \mathbb{R}^{d \times n}$ and a target vector $\mathbf{t} \in \mathbb{R}^d$, find the lattice point $\mathbf{Bx}$ (with $\mathbf{x} \in \mathbb{Z}^n$) such that $\|\mathbf{t} - \mathbf{Bx}\|$ is minimized. This is an optimization (minimization) problem with feasible solutions given by all integer vectors $\mathbf{x} \in \mathbb{Z}^n$, and objective function $f(\mathbf{x}) = \|\mathbf{t} - \mathbf{Bx}\|$.

Let $\mathbf{B} = [\mathbf{B}', \mathbf{b}]$ and $\mathbf{x} = (\mathbf{x}', x)$, where $\mathbf{B}' \in \mathbb{R}^{d \times (n-1)}$, $\mathbf{b} \in \mathbb{R}^d$, $\mathbf{x}' \in \mathbb{Z}^{n-1}$ and $x \in \mathbb{Z}$. Notice that if you fix the value of x, then the CVP problem $(\mathbf{B}, \mathbf{t})$ asks to find the value $\mathbf{x}' \in \mathbb{Z}^{n-1}$ such that

$$\|\mathbf{t} - (\mathbf{B}'\mathbf{x}' + \mathbf{b}x)\| = \|(\mathbf{t} - \mathbf{b}x) - \mathbf{B}'\mathbf{x}'\|$$

is minimized. This is also a CVP instance $(\mathbf{B}', \mathbf{t}')$ with a modified target $\mathbf{t}' = \mathbf{t} - \mathbf{b}x$, and a lower dimensional lattice $\mathcal{L}(\mathbf{B}')$. In particular, the solution space now consists of an $(n-1)$ integer variables $\mathbf{x}'$. This suggests one can solve CVP by setting the value of $\mathbf{x}$ one coordinate at a time.

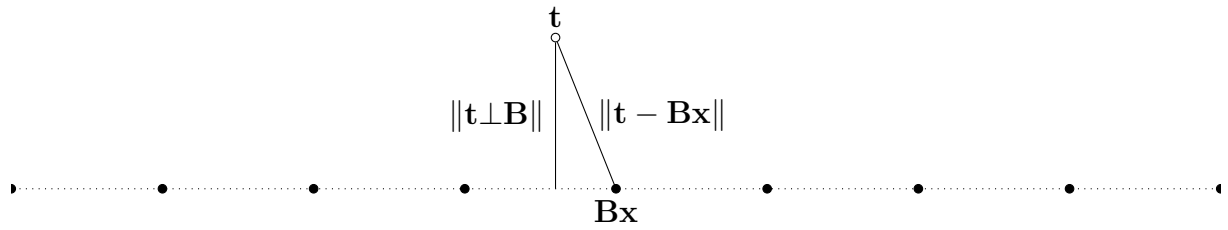
There are several ways one can turn this dimension reduction approach into an algorithm, using several standard algorithmic design techniques. The simplest techniques are

- the *greedy* method, which results in approximate solutions, but runs in polynomial time, and
- *branch and bound*, which leads to exact solutions in superexponential time.

Both of them are based on a very simple lower bound on the objective function:

$$\min_{\mathbf{x}} f(\mathbf{x}) = \mathrm{dist}(\mathbf{t}, \mathcal{L}(\mathbf{B})) \geq \mathrm{dist}(\mathbf{t}, \mathrm{span}(\mathbf{B})) = \|\mathbf{t} \perp \mathbf{B}\|$$

The distance of a target $\mathbf{t}$ to the lattice is at least as big as $\|\mathbf{t} \perp \mathbf{B}\|$, i.e., the distance of the target to the linear span of the lattice. This trivial lower bound seems pretty uninteresting by itself, but, as we will see, it provides very useful information when used in conjunction with the dimension reduction approach outlined earlier.



2.1 Greedy Method: Babai's Nearest Plane Algorithm

The greedy method consists of choosing the variables definining the solution space one at a time, by picking each time the value that looks more promising. In our case, one chooses the value of x that gives the lowest possible value for the lower bound $\|\mathbf{t}'\perp\mathbf{B}'\|$. Recall that $\mathbf{B} = [\mathbf{B}', \mathbf{b}]$ and $\mathbf{x} = (\mathbf{x}', x)$, and that for any fixed value of x, the CVP instance $(\mathbf{B}, \mathbf{t})$ reduces to the CVP instance $(\mathbf{B}', \mathbf{t}')$ where $\mathbf{t}' = \mathbf{t} - \mathbf{b}x$. Using $\|\mathbf{t}'\perp\mathbf{B}'\|$ for the lower bound, we want to choose the value of x such that

$$\|\mathbf{t}'\perp\mathbf{B}'\| = \|\mathbf{t} - \mathbf{b}x\perp\mathbf{B}'\| = \|(\mathbf{t}\perp\mathbf{B}') - (\mathbf{b}\perp\mathbf{B}')x\|$$

is as small as possible. This is a very easy 1-dimensional CVP problem (with lattice $\mathcal{L}(\mathbf{b}\perp\mathbf{B}')$ and target $\mathbf{t}\perp\mathbf{B}'$) which can be immediately solved setting

$$x = \left\lfloor \frac{\langle \mathbf{t}, \mathbf{b}^* \rangle}{\|\mathbf{b}^*\|^2} \right\rceil$$

where $\mathbf{b}^* = \mathbf{b}\perp\mathbf{B}'$ is the component of $\mathbf{b}$ orthogonal to the other basis vectors. That's it! The full algorithm is given below:

$\textsc{Greedy}([],\mathbf{t}) = \mathbf{0}$
$\textsc{Greedy}([\mathbf{B}, \mathbf{b}],\mathbf{t}) = c \cdot \mathbf{b} + \textsc{Greedy}(\mathbf{B},\mathbf{t} - c \cdot \mathbf{b})$
where $\mathbf{b}^* = \mathbf{b}\perp\mathbf{B}$
$x = \langle \mathbf{t}, \mathbf{b}^* \rangle / \langle \mathbf{b}^*, \mathbf{b}^* \rangle$
$c = \lceil x \rfloor$

It is easy to check that the algorithm always return a lattice vector, and that it performs a polynomial number of arithmetic operations. The size of all numbers involved is also easy to bound using the fact that all Gram-Schmidt orthogonalized vectors $\mathbf{b}^*$ are polynomial time computable. The interesting part is to bound the quality of the output.

Exercise 1 *Prove that for any lattice basis* $\mathbf{B}$ *and target vector* $\mathbf{t} \in span(\mathbf{B})$*, the algorithm* Greedy *returns a lattice vector* $\mathbf{v} \in \mathcal{L}(\mathbf{B})$ *such that* $\mathbf{t} \in \mathbf{v} + \mathcal{C}(\mathbf{B}^*)$*, where* $\mathbf{B}^* = [\mathbf{b}_1^*, \ldots, \mathbf{b}_n^*]$ *is the Gram-Schmidt orthogonalization of the basis matrix* $\mathbf{B}$*. In particular,* $\|\mathbf{t} - \mathbf{x}\| \leq \frac{1}{2}\sqrt{\sum_i \|\mathbf{b}_i^*\|^2}$.

The exercise above provides an absolute[2] bound on the distance between the target and the lattice point returned by the algorithm. However, it does not solve CVP_γ: if $\mathbf{B}$ is an arbitrary lattice basis, the greedy algorithm can return solutions that are arbitrarily worse than the best possible.

Exercise 2 *Prove that for any γ (possibly a function of n), there is an input $(\mathbf{B}, \mathbf{t})$ such that the Greedy algorithm returns a lattice point at distance $\gamma \cdot \mu$ from $\mathbf{t}$, where μ is the distance of $\mathbf{t}$ to the closest lattice point.* [Hint: you can find bad inputs in dimension as low as 2.]

Sometime the greedy algorithm is used to solve a different variant of CVP, called the *Bounded Distance Decoding* problem (BDD). This is the problem of finding the lattice point closest to a target $\mathbf{t}$, under the promise that $\mathbf{t}$ is very close to the lattice. (Typically, closer than $\lambda/2$.) The next exercise shows that the greedy algorithm solves BDD, but again within a radius that depends on the input basis.

Exercise 3 *Show that if $dist(\mathbf{t}, \mathcal{L}(\mathbf{B})) < \frac{1}{2} \min_i \|\mathbf{b}_i^*\|$, then* Greedy*($\mathbf{B}$, $\mathbf{t}$) returns the (necessarily unique) lattice point closest to $\mathbf{t}$.*

Combining the two exercises together, one can show that Greedy solves the approximate CVP_γ within an approximation factor $\gamma = \sqrt{\sum_i \|\mathbf{b}_i^*\|^2} / \min_i \|\mathbf{b}_i\|$.

Theorem 2 *Let $\mathbf{B}$ be a lattice basis and let $\mathbf{B}^*$ be its Gram-Schmidt ortogonalization. On input $\mathbf{B}$ and any target vector $\mathbf{t}$, the Greedy algorithm finds a lattice vector $\mathbf{v} \in \mathbf{B}\mathbb{Z}^n$ such that $\|\mathbf{t} - \mathbf{v}\| \leq \gamma \cdot \mu$, where μ is the distance of $\mathbf{t}$ from the closest lattice point, and $\gamma = \max_j \sqrt{\sum_{i \leq j} (\|\mathbf{b}_i^*\| / \|\mathbf{b}_j^*\|)^2}$.*

The bound γ in the quality of the CVP approximation achieved by the Greedy algorithm depends in the basis $\mathbf{B}$, and more specifically, on the length of the orthogonalized vectors $\|\mathbf{b}_i^*\|$. We have already seen that the approximation factor γ can be arbitrarily bad if the basis $\mathbf{B}$ is not properly chosen. The bound in the theorem suggests that a good basis is one such that all $\mathbf{b}_i^*$ are roughly the same. Notice that the product of the $\|\mathbf{b}_i^*\|$ is constant because it equals the determinant of the lattice. So, if some of them is much smaller than $\det(\Lambda)^{1/n}$, then some other will be bigger, resulting in poor approximation factor. In the next lecture, we will study "basis reduction" algorithms that allow to compute lattice bases which, used in conjuction to the Greedy algorithm, allow to solve CVP_γ within a factor γ that depends only on the lattice dimension. We anticipate that using the LLL basis reduction algorithm, the Greedy algorithm solves CVP_γ in polynomial time for $\gamma = 2^n$.

Exercise 4 *Show that a large $\max \|\mathbf{B}^*\| / \min \|\mathbf{B}^*\|$ gap does not necessarily mean* Greedy *produces poor* CVP *approximations. More specifically, show that for any (arbitrarily large) $\gamma > 1$ there is a lattice basis $\mathbf{B}$ such that $\max \|\mathbf{B}^*\| / \min \|\mathbf{B}^*\| > \gamma$, and still nounGreedy($\mathbf{B}, \mathbf{t}$) solves* CVP *exactly for any target $\mathbf{t}$.*

[2] By "absolute" we mean that it depends only on the basis, and not on the target vector.

As a historical node, the above Greedy algorithm to approximate CVP using LLL reduced bases was originally proposed by Babai, and therefore it is often referred to as Babai's algorithm. Another common name for the algorithm is the *nearest plane* algorithm because of the following natural geometric interpretation: at each step, the algorithm choose the value of c corresponding to the hyperplace $c\mathbf{b} + \mathcal{L}(\mathbf{B})$ nearest to the target.

2.2 Branch and Bound: The Enumeration Algorithm

The greedy algorithm runs in polynomial time, but only provides approximate solutions to CVP. Sometime, one needs to solve CVP exactly, i.e., find a lattice point which is closest to the target. This can be done using the same overall framework as used in the Greedy algorithm, but trying several possible values for each variable. More specifically, rather than setting x_n greedily on the most promising value (i.e., the one for which the distance lower bound $\|\mathbf{t}'\perp\mathbf{B}'$ is minimal), we bound the set of all possible values that x can take, and then we branch on all of them to solve each corresponding subproblem independently. To conclude, we select the best possible solution among those returned by all branches.

In order to bound the values that x can take, we also need an upper bound on the distance of the target to the lattice. This can be obtained in several ways. For example, one could simply use $\|\mathbf{t}\|$ (the distance of the target to the origin) as upper bound. Typically better, one can use the Greedy algorithm to first find an approximate solution $\mathbf{v} = \textsc{Greedy}(\mathbf{B}, \mathbf{t})$, and use $\|\mathbf{t} - \mathbf{v}\|$ as the upper bound. Once an upper bound u has been established, one can restrict variable x to those values such that $\|(\mathbf{t} - x\mathbf{b})\perp\mathbf{B}'\| \leq u$.

The resulting algorithm is similar to the greedy one, and it is described below:

$\textsc{Branch\&Bound}([],\mathbf{t}) = \mathbf{0}$
$\textsc{Branch\&Bound}([\mathbf{B}, \mathbf{b}],\mathbf{t}) = \text{closest}(V,\mathbf{t})$
where $\mathbf{b}^* = \mathbf{b}\perp\mathbf{B}$
$\mathbf{v} = \textsc{Greedy}(\mathbf{B},\mathbf{t})$
$X = \{x\colon \|(\mathbf{t} - x\mathbf{b})\perp\mathbf{B}\| \leq \|\mathbf{t} - \mathbf{v}\|\}$
$V = \{x \cdot \mathbf{b} + \textsc{Branch\&Bound}(\mathbf{B}, \mathbf{t} - x \cdot \mathbf{b})\colon x \in X\}$

where closest(V,t) selects the vector in $V \subset \mathcal{L}(\mathbf{B})$ closest to the target $\mathbf{t}$.

As for the Greedy algorithm, the performance (in this case, the running time) of the Branch&Bound algorithm can be arbitrarily bad, if we first don't reduce the input basis.

Theorem 3 *The total number of recursive calls performed by the Branch&Bound algorithm on input a basis* $\mathbf{B}$ *and target* $\mathbf{t}$ *is at most* $T = \prod_i \left\lceil \sqrt{\sum_{i \leq j} (\|\mathbf{b}_i^*\|/\|\mathbf{b}_j^*\|)^2} \right\rceil$.

Variants of the branch and bound algorithm were proposed by several authors, including Fincke and Pohst, Kannan, Schnorr and Euchner. Different variants are often referred to by quite different names, like the *sphere decoder* or *enumeration algorithm*, all justified by the natural geometric interpretation of the algorithm which enumerates all lattice points within a sphere around the target vector. The difference between different variants of the

algorithms is often just in the preprocessing of the lattice basis (e.g., whether it is LLL reduced before running the main algorithm), different choices in the selection of the upper bound function $u(\mathbf{B}, \mathbf{t})$, the order in which the elements of $x \in X$ are processed, or just low level implementation details. (E.g., the algorithm is more typically described using nested loops rather than recursion, it may employ floating point numbers to speed up the Gram-Schmidt computation, etc.) Two interesting variants that are worth a special mention are the enumeration algorithm of Schnorr and Euchner, which is the most commonly used in cryptanalysis, and reported to outperform most other methods in practice, and the algorithm of Kannan, which is the currently best polynomial space algorithm for the exact solution of CVP.

The main ideas in the algorithm of Schnorr and Euchnerr are to process the possible values of x in order of nondecreasing value of $\|\mathbf{t}'\perp\mathbf{B}'\|$, and to dynamically update the upper bound u to the distance of $\mathbf{t}$ to the closest lattice vector found so far. In a sense, this is a hybrid between the greedy and branch and bound solutions, where the most promising value of x is selected first, but then other values x are considered as well (in an order that reflects how promising they are). Notice that since the first lattice vector found by this enumeration strategy is precisely the one returned by the greedy algorithm, there is no need to explicitly run the greedy algorithm to compute an upper bound on the distance. The algorithm considers all possible values of x in increasing order of $\|\mathbf{t} - \mathbf{b}x\perp\mathbf{B}'\|$, and keeps track of the closest lattice vector $\mathbf{v}$ found after each iteration. The loop terminates when $\|\mathbf{t} - \mathbf{b}x\perp\mathbf{B}'\| \geq \|\mathbf{t} - \mathbf{v}\|$.

The main idea in Kannan's algorithm is to invest much more time preprocessing the basis $\mathbf{B}$ in order to speed up the actual branch and bound enumeration procedure. More specifically, Kannan first computes a basis such that $\mathbf{b}_1$ is a shortest nonzero vector in the lattice. This is done by first recursively preprocessing the (projection orthogonal to $\mathbf{b}_1$ of) rest of the basis $[\mathbf{b}_2, \ldots, \mathbf{b}_n]$, and then finding the shortest vector using a variant of the enumeration algorithm. While the recursive preprocessing proposed by Kannan involves a substantial overhead (making it noncompetitive in practice against other variants of the algorithm), it has the theoretical advantage of substantially reducing the asymptotic running time of the overall enumeration procedure form $2^{O(n^2)}$ down to $2^{O(n \log n)} = n^{O(n)}$. Unfortunately, the overhead is such that Kannan's algorithm is outperformed in practice by other asymptotically inferior enumeration strategies with worst-case running time $2^{O(n^2)}$. There are also other (not enumeration based) algorithms that are asymptotically even faster than Kannan, with running time $2^{O(n)}$, but again not competitive in practice.