6.S897 Algebra and Computation March 11, 2005

Lecture 11

Lecturer: Madhu Sudan *Scribe: Alex Wein*

Today

- Gauss and LLL lattice algorithms
- Summary of factorization

1 Shortest Vector Problem

Definition 1 (Shortest Vector Problem) *Given $v_1, \ldots, v_k \in \mathbb{Z}^k$, find $\alpha_1, \ldots, \alpha_k \in \mathbb{Z}$ (not all zero) to approximately minimize $\| \sum_i \alpha_i v_i \|_2$ in the following sense: if $\exists \{\tilde{\alpha}_i\}_{i=1}^k$ such that $\| \sum_i \tilde{\alpha}_i v_i \| < \beta$ then we will find $\{\alpha_i\}$ such that $\| \sum_i \alpha_i v_i \| \leq \gamma(k) \cdot \beta$.*

We will see two algorithms that solve this problem. Gauss' algorithm solves the $k = 2$ case exactly, i.e. $\gamma(k) = 1$. The LLL algorithm solves the general case (arbitrary k) with $\gamma(k) = 2^k$.

1.1 Gauss Algorithm

The algorithm will maintain a basis consisting of a "small" vector $s \in \mathbb{Z}^2$ and a "big" vector $b \in \mathbb{Z}^2$. The idea is to subtract multiples of s off of b in order to make b smaller. Then s and b swap roles and we repeat. Formally, the algorithm proceeds as follows.

Gauss Algorithm

1. Input: $s, b \in \mathbb{Z}^2$
2. $b \leftarrow b - is$ where i minimizes $\|b - is\|$
3. If b is **sufficiently small** then swap(b, s) and go to step 2; otherwise stop

To complete the description of the algorithm, we need to specify what is meant by *sufficiently small*. Imagine viewing the vectors s and b in the plane, with s aligned with the positive horizontal axis. Draw an axis-aligned square, centered at the origin with side-length $\|s\|$ (so that the square passes through $s/2$). We will say that b is *sufficiently small* if it lies in this square. Note that step 2 ensures that the projection of b onto s lies between $-s/2$ and $s/2$, i.e. the horizontal coordinate of b lies within the square. If the vertical coordinate of b

also lies within the square then the algorithm terminates; otherwise we swap b and s and continue.

To see that the algorithm terminates, note that at each iteration, the length of s decreases by a factor of $1/\sqrt{2}$ because any vector inside the square has length at most $\|s\|/\sqrt{2}$. All that remains is to show the correctness of the algorithm.

Claim 2 *When Gauss' algorithm terminates, either b or s is the shortest (nonzero) vector in the lattice.*

Proof Let $v = is + jb$ be the true shortest vector, where $i, j \in \mathbb{Z}$. If $j = 0$ then $\|v\| \geq \|s\|$ which means s is the shortest vector and we are done. Otherwise, write $b = b^* + \alpha s$ with $-\frac{1}{2} \leq \alpha \leq \frac{1}{2}$, $b^* \in \mathbb{R}^2$, and $b^* \perp s$. The termination condition implies that b lies outside the square and so $\|b^*\| \geq \frac{1}{2}\|s\|$. If $j \geq 2$ then again $\|v\| \geq \|s\|$ and we are done. Consider the remaining case, $j = 1$. We have $\|b\|^2 = \|b^*\|^2 + \alpha^2\|s\|^2 \leq \|b^*\|^2 + (i+\alpha)^2\|s\|^2 = \|b + is\|^2 = \|v\|^2$ using the fact that $|\alpha| \leq \frac{1}{2}$ implies $|i + \alpha| \geq |\alpha|$ for any $i \in \mathbb{Z}$. This shows $\|b\| \leq \|v\|$ so we are done.

1.2 LLL Algorithm

Now we present the LLL (Lenstra-Lenstra-Lovász) lattice basis reduction algorithm, which finds a 2^k-approximation to the shortest vector in any number of dimensions k. The basic outline of the algorithm is as follows.

LLL Algorithm

1. Input: $b_1, \ldots, b_k \in \mathbb{Z}^k$
2. **Orthogonalize** $b_1, \ldots, b_k$
3. Find some i for which $\|b_{i+1}^*\| \leq \frac{1}{2}\|b_i^*\|$, swap$(b_i, b_{i+1})$ and go to step 2; if no such i exists then stop and output b_1

In the following, we will clarify what is meant by *orthogonalize*, and we will define the b_i^*'s mentioned in step 3. The idea of the orthogonalization step is to subtract copies of b_i's from one another in order to get an approximately orthogonal basis for the lattice. This is done in a similar manner to the Gram-Schmidt process, except we can only subtract integer multiples of one vector from another so that we stay in the lattice. Formally, for $i = 1, \ldots, k$, let $b_i^* \in \mathbb{R}^k$ be the projection of b_i orthogonal to span$(b_1, \ldots, b_{i-1})$ so that $b_i^* \perp$ span$(b_1, \ldots, b_{i-1})$. Note that $\{b_i^*\}$ are orthogonal but do not necessarily lie in the lattice. For each i we can write $b_i^* = b_i + \sum_{j<i} \mu_{ij} b_j$ for some scalars $\mu_{ij} \in \mathbb{R}$. The orthogonalization step (step 2 of the algorithm) is to update the b_i's according to $b_i \leftarrow b_i + \sum_{j<i} [[\mu_{ij}]] b_j$ where $[[\cdot]]$ denotes rounding to the nearest integer. Although the algorithm does not explicitly use the b_i^*'s or μ_{ij}'s, for purposes of analysis we keep the b_i^*'s unchanged but update the μ_{ij}'s according to $\mu_{ij} \leftarrow \mu_{ij} - [[\mu_{ij}]]$ so that the relation $b_i^* = b_i + \sum_{j<i} \mu_{ij} b_j$ still holds. Note that the new μ_{ij}'s satisfy $|\mu_{ij}| \leq \frac{1}{2}$.

Next we discuss step 3 of the algorithm. The idea is to swap pairs (b_i, b_{i+1}) in order to bring shorter vectors closer to the front (i.e. to lower indices). The condition $\|b_{i+1}^*\| \leq \frac{1}{2}\|b_i^*\|$ is analagous to the swap condition for Gauss' algorithm (check if b is inside the square). After we swap a single pair we redefine the b_i^*'s (and μ_{ij}'s) according to their definition, i.e. b_i^* is the projection

of the new b_i orthogonal to the span of the new $b_1, \ldots, b_{i-1}$. Note however, that swapping (b_i, b_{i+1}), only changes b_i^* and b_{i+1}^* (and leaves the other b_j^*'s unchanged). After performing a single swap (b_i, b_{i+1}) the algorithm returns to the orthogonalization step (step 2).

The analysis of the LLL algorithm has two components. We need to show that the algorithm terminates after a polynomial number of iterations, and we need to prove the 2^k approximation guarantee.

Claim 3 *The LLL algorithm terminates after a polynomial number of iterations.*

Proof Define the potential function $\Phi = \prod_{i=1}^{k} \Phi_i$ where $\Phi_i = \prod_{j \le i} \|b_j^*\|$. Equivalently, $\Phi_i = |\det(b_1, \ldots, b_i)|$, which is the i-dimensional volume enclosed by the vectors $b_1, \ldots, b_i$. When we swap (b_i, b_{i+1}), Φ_i changes but the remaining Φ_j's are unchanged; this is because the only b_j^*'s that change are b_i^* and b_{i+1}^*, and the product $\|b_i^*\| \cdot \|b_{i+1}^*\|$ is unchanged (since by the volume interpretation, Φ_{i+1} is unchanged). Furthermore, we will show that Φ_i decreases by a constant factor, similarly to the analysis of Gauss' algorithm. The implies the claim because the initial value of Φ is only exponential in the size of the input. Let b_j^* denote the variables before swapping (b_i, b_{i+1}), and let $\tilde{b}_j^*$ denote the variables after swapping. We need to show that $\tilde{b}_i^*$ is shorter than b_i^* by a constant factor. The swap procedure and definition of the b_j^*'s ensure that $\tilde{b}_i^*$ is the projection of b_{i+1} orthogonal to the span of $b_1, \ldots, b_{i-1}$. Start with the equation $b_{i+1}^* = b_{i+1} + \sum_{j<i+1} \mu_{i+1,j} b_j$ and project both sides orthogonal to $b_1, \ldots, b_{i-1}$ to get $b_{i+1}^* = \tilde{b}_i^* + \mu_{i+1,i} b_i^*$ and so $\tilde{b}_i^* = b_{i+1}^* - \mu_{i+1,i} b_i^*$. Since $b_{i+1}^* \perp b_i^*$, $\|b_{i+1}^*\| \le \frac{1}{2}\|b_i^*\|$ (by the swapping criterion), and $|\mu_{i+1,i}| \le \frac{1}{2}$ (by the orthogonalization step), we have $\|\tilde{b}_i^*\| \le \|b_i^*\|/\sqrt{2}$. At each iteration, Φ_i decreases by a factor of $1/\sqrt{2}$ and so Φ also decreases by this factor.

Claim 4 *The LLL algorithm outputs a 2^k-approximation of the shortest (nonzero) vector in the lattice.*

Proof Consider $\{b_i\}$ and $\{b_i^*\}$ upon termination and write the true shortest vector v as an integer combination $v = \sum_{i=1}^{k} \alpha_i b_i$ with $\alpha_i \in \mathbb{Z}$. Find the largest j for which $\alpha_j \neq 0$. The swapping criterion implies $\|b_{i+1}^*\| \ge \frac{1}{2}\|b_i^*\|$ for all i and so by induction, $\|b_j^*\| \ge 2^{-j}\|b_1^*\|$. Now we have $\|v\| = \|\sum \alpha_i b_i\| \ge \|\alpha_j b_j^*\| \ge \|b_j^*\| \ge 2^{-j}\|b_1^*\| = 2^{-j}\|b - 1\| \ge 2^{-k}\|b_1\|$ and so $\|b_1\| \le 2^k \|v\|$.

2 Summary of Factorization

We have seen how to factor over $\mathbb{F}_p[x]$ and $\mathbb{Q}[x]$. Two natural ways to take a field $\mathbb{F}$ and construct a bigger field are $\mathbb{F} \to \mathbb{F}(y)$ (e.g. bivariate polynomials) and $\mathbb{F} \to \mathbb{F}[y]/g(y)$ where g is irreducible (e.g. finite fields $\mathbb{F}_q$). We have seen via Hensel lifting that if you can factor over $\mathbb{F}$ then you can factor over $\mathbb{F}(y)$.

2.1 Factoring over $\mathbb{F}[y]/g(y)$

We can also show that if you can factor over some field $\mathbb{F}$ then you can factor over $\mathbb{F}[y]/g(y)$. Suppose we want to factor $Q(x, y) = A(x, y)B(x, y) \pmod{g(y)}$ where A, B are unknown. We want to reduce the problem to factoring a different polynomial $q(x) \in \mathbb{F}[x]$. By assumption we know how to factor $q(x) = a(x)b(x)$

and we want the factors a, b to tell us something about the factors A, B of Q. The idea is to take resultants of Q, A, B with respect to y. The resultant $R_Q(x)$ generates the ideal $(Q, g) \cap (x)$, and similarly $R_A(x)$ and $R_B(x)$ generate $(A, g) \cap (x)$ and $(B, g) \cap (x)$ respectively. Here the purpose of intersecting with (x) is to only consider polynomials that are purely polynomials in x (and not y). Assuming $R_A(x)$ and $R_B(x)$ are relatively prime, $R_Q(x) = R_A(x)R_B(x)$. Therefore, given $Q(x, y)$ we can factor $q(x) \equiv R_Q(x)$ to recover $a(x) \equiv R_A(x)$ and $b(x) \equiv R_B(x)$. Then let $A(x, y) = \gcd(R_A(x), Q(x, y))$ and $B(x, y) = \gcd(R_B(x), Q(x, y))$ to recover the factorization $A(x, y)B(x, y)$ of Q.

2.2 Factoring in n Variables

The conclusion from the above is that (informally) we know how to factor over any field that we can describe to a computer. (Note that of course we can't factor over the ring of integers $\mathbb{Z}$.) Namely, we can factor over any field that can be obtained from $\mathbb{F}_p$ or $\mathbb{Q}$ by a finite number of extensions of the form $\mathbb{F} \to \mathbb{F}(y)$ and/or $\mathbb{F} \to \mathbb{F}[y]/g(y)$. The one caveat is that the runtime degrades with each extension, so if we want a polynomial-time algorithm, we can only take a constant number of extensions. This begs the question of whether we can factor a polynomial $f \in \mathbb{Q}[x_1, \ldots, x_n]$ of degree n, in time poly(n). There is one potential issue with this question, which is that such a polynomial f might have an exponential number of nonzero coefficients. And even if f is sparse (few nonzero coefficients), one of its factors might not be. However, we can avoid these problems as follows. Suppose we have a black-box procedure that can compute f on any input $(\alpha_1, \ldots, \alpha_n)$. Then there is indeed an algorithm to factor f in time poly(n). Similarly to the input, each factor g in the output is described succinctly as a procedure $\mathcal{P}_g$ that can evaluate g on any $(\alpha_1, \ldots, \alpha_n)$. The procedure $\mathcal{P}_g$ is allowed to make calls to the original black box for f. The key ingredient in this factorization algorithm is Hilbert's irreducibility. The main idea is to look at f on a 2-d surface, i.e. perform the change of variables $x_i \leftarrow \alpha_i\theta + \beta_i\gamma + \delta_i$ for random constants $\alpha_i, \beta_i, \gamma_i$ to get a problem in only two variables θ, γ. With high probability, reducing to two variables preserves irreducibility of factors. When $\mathcal{P}_g$ is asked to evaluate g on a point α that is not on the 2-d surface, it considers the 3-d surface containing the 2-d surface and α.