On Quantum Sieve Approaches to the Lattice Shortest Vector Problem

Daniel Epelbaum

December 2014

1 Introduction

The lattice hortest vector problem, or lattice SVP, has gained a lot of attention in the field of quantum computing. There are a number of reasons for this, including the fact that the hardness of lattice SVP is the foudation of a number of post-quantum cryptosystems and that approximate-SVP is in $NP \cap coNP$, so it is a prime target to use in searching for exponential speedups over classical algorithms [5, 7, 4].

In this paper, we are interested in a particular attempt at a quantum algorithm for lattice SVP. In 2003 Regev showed a polynomial-time quantum reduction to the dihedral coset problem, a variant of the dihedral hidden subgroup problem [5]. In 2008, Kuperberg devised a subexponential algorithm for DCP [3], but this in conjunction with Regev's reduction did not lead to a subexponential algorithm for lattice SVP. On one level this is because Kuperberg's algorithm runs in $2^{O(\sqrt{n})}$ time, and Regev's reduction gives a quadratic blowup in the input size. This paper tries to give a deeper reason why Kuperberg's algorithm does not work to provide a subexponential algorithm for lattice SVP. Essentially the reason is that there exist classical sieves for lattice SVP that are too similar, and so the quantum sieve cannot really give a substantial improvement. We will see more specifically what this means.

2 An Overview of Lattices and Definition of Lattice SVP

A *lattice* is a discrete additive subgroup of R^m. A *basis* for a lattice L is a set of n linearly independent vectors $b_1...b_n$ s.t. $L = \{\Sigma_{i=1}^{n} a_i b_i | \forall i \ a_i \in Z\}$, i.e. a set of vectors s.t. the lattice is the set of all integer combinations of these vectors. If we are given a set of linearly independent vectors $B = b_1...b_n$, we denote the lattice they generate by $L(B)$. It is true, though we don't prove it here, that every lattice L in R^m has a basis of n vectors with $n \leq m$. In the case $n = m$, we say that L is a full-rank lattice. We restrict attention to full-rank lattices, as we can always reduce the dimension of our space to be the dimension of the lattice in a polynomial time by looking at the space spanned by the lattice vectors.

We denote by $sh(L)$ the norm* of the shortest non-zero vector in L. We can now define SVP and $f(n)$-approximate SVP.

Definition 2.1. *The* lattice shortest vector problem *or* lattice SVP *takes as input a basis B of n linearly independent vectors and outputs a vector $u \in L = L(B)$ with $||u|| = sh(L)$.*

Similarly we have:

Definition 2.2. *The* $f(n)$-approximate lattice shortest vector problem *or* $f(n)$-approximate SVP *takes as input a basis B of n linearly independent vectors and outputs a vector $u \in L = L(B)$ with $||u|| \leq f(n) * sh(L)$.*

In the latter problem we will mainly investigate the cases when $f(n) = poly(n)$. This is for two reasons: this level of approximation is enough to invalidate the security assumptions of a number of lattice based cryptosystems[7] , and there already exist polynomial-time algorithms for the case when $f(n) = exp(n)$. In particular we have:

Theorem 2.3. *Given a basis B of n vectors, there exists a basis for $L(B)$ denoted by $\overline{B}$, called an* LLL-reduced basis*, with the property that $||\overline{b_1}|| \leq 2^n * sh(L(B))$. Furthermore, such a basis can be found in time polynomial in n.*

A discussion of this can be found in [5, 4]. Then we have an algorithm for 2^n-approximate SVP: given a basis B, we find $\overline{B}$, which we can do in polynomial time, and we output $\overline{b_1}$. This algorithm is important for a number of reasons, amongst them the fact that both the classical and quantum sieves start by finding such a basis, and the complexity analysis of both of these algorithms relies on theorem 2.3. Furthermore, because such a basis is computable from any

*the norm will be denoted $||v||$ and will refer to the L_2-norm

other basis, showing that either SVP or its approximate variants requires superpolynomial or exponential time can be simplified to showing that SVP (or approximate variants) is hard starting from an LLL-reduced basis. In other words, because we are looking at algorithms that take more than polynomial time, we may as well begin by reducing our basis to an LLL-reduced basis.

3 Lattice SVP: Classical Sieve

We now present the classical randomized algorithm for lattice SVP that requires exponential time. This algorithm is due to Ajtai, Kumar and Sivakumar and we will refer to it as the AKS algorithm or the AKS sieve. The algorithm involves a number of parameters and a complete formal description is beyond the scope of this paper, but can be found in [1]. We present here a simplified version of the algorithm that accounts for its exponential time requirement. A more thourough analysis is required to find the constant factor in the exponent, and such analysis can be found in [4] on which the present description is closely based. The existence and complexity of the algorithm is given by the following theorem:

Theorem 3.1. *There exists a randomized algorithm that solves SVP and fails with exponentially small probability.*

Proof. (sketch) The algorithm works essentially by finding an r so that a ball B_r centered at 0 with radius r contains $2^{O(n)}$ lattice points, so that taking $2^{\Omega(n)}$ samples of lattice points in this ball gives a high probability that one of these vectors is the shortest vector. The value for such an r is given by the following lemma:

Lemma 3.2. *Given a lattice L and a ball B_r of radius r, $|L \cap B_r| \leq (1 + \frac{2r}{sh(L)})^n$.*

To prove this lemma, you simply note that if you take balls of radius less than $sh(L)/2$ around each point in the larger ball, each ball contains a single lattice point. Then the number of lattice points in B_r is bounded by the volume of B_r divided by the volume of a smaller such ball. This gives the desired expression.

Then we want to find an $r = O(sh(L))$. This value can be guessed, precisely because of the LLL-reduction algorithm. To see this, note that is is sufficient to find an r such that $sh(L) \leq r \leq 2 * sh(L)$. Then for an LLL-reduced basis B, we have $||b_1|| \leq 2^n * sh(L)$ and so there are n possible values of a constant c so that $r = 2^{-c} * ||b_1||$. Then we can simply try our algorithm with each value of c until we get the correct one. This adds at most a factor of n to our runtime. Then we need a way to find $2^{\Omega(n)}$ samples in B_r. It is not clear how to do this naively. This is where the sieve comes into play. To begin with, we have to make use of an algorithm known as Babai's Rounding which gives the following lemma:

Lemma 3.3. *Given an LLL-reduced basis B, and some vector t in R^n, there exists a polynomial time algorithm that outputs a vector v in $L(B)$ s.t. $||v - t|| \leq \Sigma_{i=1}^{n} \frac{||b_i||}{2}$. In particular this means that $||v - t|| \leq c^n * sh(L(B))$ for some constant c.*

The proof of this lemma is not relevant to our analysis, so we leave it out. For more details see[1, 4]. We can use this algorithm to create samples of L in B_{r_0} for $r_0 = 2^{O(r)}$ by sampling a number of $O(1)$-length vectors in R^n and applying Babai's Rounding. Then we want to show that by using a sieve we can get lattice points in B_r from points in B_{r0}. The sieve itself works essentially as follows: we start with the given sample points, and we create a set of "representatives" which is a subset of our points such that each point is at most a distance $\frac{r_0}{2}$ away from a representative. We then subtract our points by their representative points, and discard the representatives. This will reduce the length of our vectors by some constant. What requires more involved analysis is that such a representative set can be found that isn't too big so that we are left with $2^{\Omega(n)}$ vectors after running the sieve. A proof of this can be found in [1]. We then simply apply the sieve so that the size of the lattice points is reduced by this constant factor enough times so that r_0 is reduced to r. Since the constant factor is bounded below by $\frac{1}{2}$, we see that we need run this sieve a linear number of times. Each time takes an exponential number of steps—it involves an exponential number of arithmetic operations after Babai rounding—and so we have our algorithm. □

4 Towards A Quantum Approach: Dihedral Hidden Subgroup Problem

We begin by defining DHSP and DCP.

Definition 4.1. *Given a group G and a black-box function f, we say f* hides *a subgroup $H \leq G$ iff f is constant on right cosets of H, and takes on different values for each distinct right coset of H. The* hidden subgroup problem *then takes as input G and f and asks to find H. The* dihedral hidden subgroup problem *is simply HSP over the dihedral group.*

Definition 4.2. *The* dihedral group of order $2N$, D_N, *is the group of symmetries of a regular N-gon in R^2. It can be seen as the quotient of the free group on 2 elements x and y with the relations $x^N = y^2 = xyxy = 1$.*

The above definitions are given largely to straighten out conventions: sometimes left cosets are used for HSP, and sometimes D_N is defined as the symmetries of a regular $\frac{N}{2}$-gon, so that the size of the group is N. For a more thorough development of HSP see [7] and for the dihedral group see [3]. We also have the following definition:

Definition 4.3. *The* dihedral coset problem *or* DCP *takes as input polynomially many registers in the state $\frac{1}{\sqrt{2}}(\mid 0, x\rangle + \mid 1, (x+d) \ mod \ N\rangle)$ where x can vary from register to register, but d is constant.*

A result of Ettinger and Høyer showed that DHSP can be reduced to the case when H is a subgroup generated by yx^d for some $d \in \{1...N-1\}$ [2]. In other words, the DHSP reduces to finding the slope of an arbitrary reflection of a regular N-gon. Informally this is because any other subgroup is generated by elements which don't contain a reflection, and so they are contained in the subgroup C_N, which is abelian. The dihedral coset problem gets its name because the inputs are registers in superpositions over cosets of such a subgroup. All known algorithms for DHSP involve a method known as "coset sampling" or simply "the standard method" which creates registers of the form involved in DCP, and then solves DCP over these registers [3].

5 The Kuperberg Algorithm for DHSP

We now turn to Kuperberg's result:

Theorem 5.1. *There exists an algorithm for the DHSP on D_N that requires only $2^{O(\sqrt{n})}$ time where $n = log(N)$ is the input length. Because the algorithm works by sampling cosets, there is a slightly modified algorithm for DCP with the same time complexity.*

Proof. Let $G = D_N$. We first show how to get from DHSP to DCP. We start by creating the uniform superposition $\frac{1}{\sqrt{|G|}} \sum_{g \in G} \mid g, f(g)\rangle$. If we then measure the registers containing $f(g)$, our state collapses to a superposition over a right coset of the hidden subgroup H, which means that for some $x \in \{1...N-1\}$ our register is in the state $\mid \psi_x\rangle = \frac{1}{\sqrt{2}}(\mid 0, x\rangle + \mid 1, (x+d) \ mod \ N\rangle)$ where d is the slope of the reflection we wish to find. By running this step a polynomial number of times, we can create the polynomially many such registers composing the input for DCP.

For the sake of comparison between quantum and classical sieves, the rest of the description of this algorithm will be divided into two parts: a reduction to an abstract classical problem, and then the solution to that problem. This analysis can be seen in [6]. Because of the nature of Regev's reduction, which we discuss below, we need only look at the case that $N = 2^n$ for some n. To begin with we describe a quantum operation on one register containing $\mid \psi_x\rangle$. To start off we apply the Quantum Fourier Transform and we get the state $\frac{1}{\sqrt{N}} \sum_{y \in Z/N} (\mid 0, y\rangle + e^{\frac{2\pi i y d}{N}} (\mid 1, y\rangle)$. By measuring y we then get $\mid \Psi_y\rangle = \frac{1}{\sqrt{2}}(\mid 0\rangle + (-1)^{\frac{2yd}{N}} \mid 1\rangle)$ for some uniformly chosen random y. The key here is that we know what y is for each register. Note that if we measured and got y in the state 2^{n-1}, then we would have the state $\mid \Psi\rangle = \mid 0\rangle + (-1)^d \mid 1\rangle$, and measuring in the Hadamard basis would tell us the parity of d. This is actually enough, because we can repeat this process replacing our black box function f with either $f_1(y^t x^s) = f(y^t x^{2s})$ if d is even, as this hides the subgroup generated by $yx^{\frac{d}{2}}$, or we call $f_2(y^t x^s) = f(y^t x^{2s+1})$, which hides the subgroup generated by $yx^{\frac{d-1}{2}}$. Repeating this process we can learn all the bits of d.

We now turn to what we can do if instead of being given only one state $\mid \Psi_y\rangle$ we have two states $\mid \Psi_{y1}\rangle$ and $\mid \Psi_{y2}\rangle$. We first create $\mid \Psi_{y1} \otimes \Psi_{y2}\rangle = \mid 0, 0\rangle + e^{2\pi i y_1 d} \mid 1, 0\rangle + e^{2\pi i y_2 d} \mid 0, 1\rangle + e^{2\pi i (y_1 + y_2) d} \mid 1, 1\rangle$. Then we apply a CNOT gate onto the right qubit controlled by the left. We measure the right qubit and on the left we have $\mid \Psi_{y1 \pm y2}\rangle = \mid 0\rangle + e^{2\pi i (y_1 \pm y_2) d} \mid 1\rangle$, with the choice $y_1 + y_2$ or $y_1 - y_2$ chosen uniformly at random and determined by the output of the measurement on the right qubit. We now have all the pieces for our classical abstraction.

Regev points out in [6] that the above can be reduced to essentially a classical problem. Suppose that we have a machine that supplies us with blocks with a random label from 1 to $2^n - 1$. Suppose that we would like to get a block with 2^{n-1}. Initially we might think we have to ask for $2^{O(n)}$ blocks before getting one with the right label. We are also, however, given a way to combine the blocks. With two blocks labeled y_1 and y_2, we can combine them and with probability $\frac{1}{2}$ we get a block labeled $y_1 - y_2$, otherwise our blocks are lost (this isn't really what happens in the above case, but if we take advantage of the possibility $y_1 + y_2$ it doesn't improve the runtime below $2^{O(\sqrt{n})}$ so we choose to just use $y_1 - y_2$). Then we can apply a sieving algorithm as follows: at the m^{th} step of the sieve we match all elements sharing mk least significant bits for some chosen constant k. We apply this until we have only the last bit left. The question we are interested in is how many blocks we need to make sure that at the end we still have at least 1. For each block of k bits we expect to have the proper matches after about $2^{O(k)}$ blocks have been produced.

If each step of the sieve combines groups of 4 blocks (since combinations have a .5 chance of failure, we expect to need to try twice) then we need $4^{O(m)} * 2^{O(k)}$ blocks. Since we have the constraint that $mk = n$, we see that we get a minimum at $m = k = \sqrt{n}$. Then we need $2^{O(\sqrt{n})}$ blocks to be able to produce a block with the correct label. If we note that producing a block in this abstraction corresponds to a polynomial number of steps in the Kuperberg sieve, then we have as a result that $2^{O(\sqrt{n})}$ is the runtime of our algorithm. □

6 Lattice SVP and DCP

There is one more piece to the puzzle: Regev's reduction. The details can be found here [5], and the rest of this section comes from this paper. The reduction itself has a very intuitive geometric motivation. Essentially, the reduction divides n-dimensional space up into regions so that there are only two lattice points in each regions. This creates what is called the "Two-Point Problem" which is the n-dimensional analogue of DCP.

Definition 6.1. *The* two-point problem *takes as input a polynomial number of registers of the form* $\frac{1}{\sqrt{2}}(|\ 0, a\rangle + |\ 1, b\rangle)$ *where* a *and* b *are elements of* $\{0...M-1\}^n$ *for some* n *and with* $M = 2^{O(n)}$*, that differ from register to register, but so that* $b - a$ *is constant across registers. The problem is to output* $b - a$*.*

An instance of the two-point problem can then be converted to an instance of DCP. It is here that an important 'blowup' occurs in the size of the problem, that will help to account for why Kuperberg's sieve does not help us to solve lattice SVP in subexponential time. The mapping is given by:

$$f(a_1...a_n) = \sum_{i=1}^{n} a_i (2M)^{(i-1)} \tag{1}$$

The mapping encodes the vector as an $O(nlog(M))$ bit string. Because $M = 2^{O(n)}$ we now have a problem with input size $O(n^2)$ bits. This blowup then explains why Kuperberg's algorithm which takes $2^{O(\sqrt{n})}$ time results in an exponential algorithm.

The reduction to the two-point problem consists of finding a function which has the property that any two lattice points that share an output must be only a distance of the shortest vector away from each other. The details of the function can be found in [5], but essentially the function works by packing balls into the space and then checking which ball the lattice point belongs to and writing that information into another register. By measuring that register, you then can collapse the state into the desired one with high probability. To deal with the problems that the grid of centerpoints might be aligned such that the lattice is lined up in a way so that each ball is empty, Regev introduces a random permutation. To deal with the fact that with small probability a register might be bad, Regev introduces the concept of the two-point problem and DCP "with failure parameter f" and argues that by creating enough registers, you can get enough successful registers. There is one final point of interest here, and that has to do with the creation of the balls. In particular, in order to get balls with the desired property—that each one contains only two lattice points—we need to have an approximation l of the norm of the shortest vector with the property that $sh(L) \leq l \leq sh(L)$. We again use LLL-reduction, as in the classical case, to obtain a $2^{(\frac{n}{2})}$-factor approximation, and then guess the $\frac{n}{2}$ possibilies for l. We develop more similarities in the next section.

7 Classical and Quantum Sieve Similarity

The rest of this paper will attempt to give a description of the commonalities between the two presented algorithms to explain why no speedup is gained in the quantum case. They both begin by performing an LLL-reduction to obtain an exponential factor approximation of the shortest vector, which they use to try all $O(n)$ possibilities for a constant factor approximation. In both cases, they then partition the space into an exponential number of regions. In the quantum case it is clear that this is what happens, but in the classical case this may not be quite as obvious. When the set of "representatives" is created, we can define "regions" that contain all of the lattice points associated with one representative. Then all of these regions are given as input to the next part of the sieve. This is analogous to the quantum case in which we create registers containing different lattice points from different regions, but in which we discard the data about which region they came from and look only at the differences between lattice points.

In the quantum case it may seem that more data is available to us sooner, in the form of the input registers to the two-point problem. If we however look at the information that is available to us, since we cannot learn a superposition, we see that it is in fact analogous to the classical case. In particular we have some number of regions. Each one can be looked at as having one lattice point at its "center" and then they look identical, and the problem

is to find the other point. There are $2^{O(n^2)}$ (the area of the region) possibilities. We of course cannot measure the first point to find this center, but we do not need to. By applying the Fourier Transform, we discard it and leave only the shift, which is what we are interested in. After we apply the Fourier transform we are still left with $2^{O(n^2)}$ possibilities. At this point we use a sieve to get through these in only exponential time in n as opposed to exponential in n^2 that would come from waiting to sample the right point. In the classical case it might seem that we have only $2^{O(n)}$ possibilities tosieve through, but this is not quite the case. The vector we output at the sieve is a sequence of differences of lattice points from our original set of $2^{O(n)}$ points. In other words, we select an exponential number of points, and we subtract an exponential number of points from another set of points an exponential number of times. If we tried enumeration, we would essentially have to try all sequences of $2^{O(n)}$ subtractions to find the output of the sieve. There are $2^{O(n^2)}$ such sequences.

In the quantum case we can in fact describe a modified version of Kuperberg's algorithm. The step that maps the two-point problem is in some ways an artifact of the way the problem was solved—first as a reduction to DCP, then as a solution to DCP. We can instead apply the Fourier Transform directly to the registers containing the two-point problem instances. The y that we get as a result is then a vector rather than a string in one dimension. The sieve we perform on the y values is then very similar to the one we perform in the classical case. We generate $2^{O(n)}$ and then match them together. In this case however instead of regions being defined by chosen "representative" points, they are defined by n-bit chunks on the n^2 bit string describing y (which can be thought of as components).

With these similarities, it is then not so surprising that the Kuperberg sieve does not provide the speedup, especially with the classical abstraction provided by Regev and discussed above. The non-classical components of quantum computation provide us with the odd machine that produces labelled blocks, but the sieve itself is in some sense "too classical."

8 Conclusion

With these similarities drawn, it then seems proper to say not that Kuperberg's sieve tries to take advantage of the similarity in structure between DCP and lattice SVP to solve the latter faster, but to solve the former faster. In other words, Kuperberg's sieve is helpful to solve DHSP and DCP faster because it uses the similarity to lattice SVP to apply a sieve that can be looked at as a quantum version of the AKS sieve. While this may be dissapointing to someone hoping to improve algorithms for lattice SVP, it does help solve DHSP faster. Kuperberg's sieve could be said to show that DHSP is not *harder* than an instance of lattice SVP of dimension $\sqrt{n}$ in some sense.

References

[1] Miklós Ajtai, Ravi Kumar, and Dandapani Sivakumar. A sieve algorithm for the shortest lattice vector problem. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 601–610. ACM, 2001.

[2] Mark Ettinger and Peter Høyer. On quantum algorithms for noncommutative hidden subgroups. *Advances in Applied Mathematics*, 25(3):239–251, 2000.

[3] Greg Kuperberg. A subexponential-time quantum algorithm for the dihedral hidden subgroup problem. *SIAM Journal on Computing*, 35(1):170–188, 2005.

[4] Phong Q Nguyen and Thomas Vidick. Sieve algorithms for the shortest vector problem are practical. *Journal of Mathematical Cryptology*, 2(2):181–207, 2008.

[5] Oded Regev. Quantum computation and lattice problems. *CoRR*, cs.DS/0304005, 2003.

[6] Oded Regev. A subexponential time algorithm for the dihedral hidden subgroup problem with polynomial space. *arXiv preprint quant-ph/0406151*, 2004.

[7] F Wang. The hidden subgroup problem (2010). *arXiv preprint quant-ph/1008.0010*.