Lenstra-Lenstra-Lovasz Algorithm

EE17B115 Adithya Swaroop

EE17B138 Nikhil M

Introduction

- Lenstra-Lenstra-Lovasz (LLL) Algorithm is an approximation algorithm of the shortest vector problem.

- It runs in polynomial time and finds an approximation within an exponential factor of the correct answer.

- It is a practical method with enough accuracy in solving integer linear programming, factorizing polynomials over integers and breaking cryptosystems.

Shortest Vector Problem

- A lattice L is a discrete subgroup generated by all the integer combinations of the vectors of some basis B.
- The Shortest vector problem (SVP) is the most famous and widely studied lattice problem, which finds the shortest non-zero vector in a given lattice L.
- The length of the vector can be defined with any norm, but most frequently with Euclidean norm.
- SVP has been studied since 19th century due to its connectivity with many problems, like integer linear programming and number theory.
- There was not efficient algorithm of solving SVP in higher dimensions until 1980s. In 1981, mathematician Perter van Emde Boas conjectured that SVP is a NP-hard problem.

Basis Reduction

Basis reduction is a process of reducing the basis of a lattice L to a shorter basis while keeping L the same. For a lattice with n vector basis we are trying to find n shortest possible vectors belonging to lattice which are linearly independent.

A 2D basis with defined to be reduced if it satisfies following condition:

$$\|\mathbf{b}_1\| \leq \|\mathbf{b}_2\|$$

$$u = \frac{\mathbf{b}_1 \cdot \mathbf{b}_2}{\|\mathbf{b}_1\|^2} \leq \frac{1}{2}.$$

This condition is ensuring the vectors are orthogonal enough. Similarly for higher dimensional reduced basis tend to have nearly orthonormal vectors

Gram-Schmidt Orthogonalization

- The idea of basis reduction in two dimensional lattice is to find the orthogonal basis based on the given basis.

- To generalize the algorithm to n-dimensions, we need to find a way to construct n-dimensional orthogonal basis based on the given basis, which leads us to Gram-Schmidt Orthogonalization.

$$\mathbf{x}_1^* = \mathbf{x}_1,$$

$$\mathbf{x}_i^* = \mathbf{x}_i - \sum_{j=1}^{i-1} \mu_{ij}\mathbf{x}_j^* \quad (2 \le i \le n), \qquad \mu_{ij} = \frac{\mathbf{x}_i \cdot \mathbf{x}_j^*}{\mathbf{x}_j^* \cdot \mathbf{x}_j^*} \quad (1 \le j < i \le n).$$

LLL basis reduction Algorithm

$k = 1$
while $k \leq n$:
  for j **from** $k-1$ **to** 0 :
    if not SizeCondition(k, j) :
      $\mathbf{b}_k = \mathbf{b}_k - \lfloor \mu_{k,j} \rceil \mathbf{b}_j$
      UpdateGramSchmidt$(\mathbf{b}_0, \ldots, \mathbf{b}_n)$
  if LovászCondition(k) :
    $k = k + 1$
  else:
    Swap$(\mathbf{b}_k, \mathbf{b}_{k-1})$
    UpdateGramSchmidt$(\mathbf{b}_0, \ldots, \mathbf{b}_n)$
    $k = \text{Max}(k-1, 1)$
return $\mathbf{b}_0, \ldots, \mathbf{b}_n$

<u>Variables</u>

$\mathbf{b}_0, \ldots, \mathbf{b}_n$

$\mathbf{b}_0^*, \ldots, \mathbf{b}_n^*$

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

Index k

Conditions in LLL Reduced basis algorithm

LLL-Reduced Basis

- Size Condition

  $$|\mu_{i,j}| \leq .5 \text{ for } 0 \leq j < i \leq n$$

- Lovász Condition

  $$|\mathbf{b}_k^*|^2 \geq (3/4 - \mu_{k,k-1})^2 \cdot |\mathbf{b}_{k-1}^*|^2 \text{ for } k = 1, \ldots, n$$

Size Condition

- Size condition checks if the kth vector is almost orthogonal to all the vectors before it. If the condition is not satisfied for vector j we can assume LLL algorithm removes the component of the jth vector from vector k.
- This operation reduces the size of the vector as short as possible and keep it as orthonormal as possible to all the k-1 vectors.
- Performing this operation for all the vectors along with Lovász Exchange Condition will make sure that our basis is reduced and approximately orthogonal which are the conditions needed to find a short basis.

Lovász Exchange Condition

$$\left\|\mathbf{b}_{i+1}^{\star} + \mu_{i+1,i}\mathbf{b}_i^{\star}\right\|^2 \geq \delta\|\mathbf{b}_i^{\star}\|^2.$$

Let us explain this mysterious condition. As Gram–Schmidt orthogonalization depends on the order of the vectors, its vectors change if $\mathbf{b}_i$ and $\mathbf{b}_{i+1}$ are swapped; in fact, only $\mathbf{b}_i^{\star}$ and $\mathbf{b}_{i+1}^{\star}$ can possibly change. And the new $\mathbf{b}_i^{\star}$ is simply $\mathbf{b}_{i+1}^{\star} + \mu_{i+1,i}\mathbf{b}_i^{\star}$; therefore, Lovász' condition means that by swapping $\mathbf{b}_i$ and $\mathbf{b}_{i+1}$, the norm of $\mathbf{b}_i^{\star}$ does not decrease too much, where the loss is quantified by δ: one cannot gain much on $\|\mathbf{b}_i^{\star}\|$ by swap.

Implementation

```
def reduction(basis, delta = 0.75) : #our main func
    n = len(basis)
    basis = list(map(Vector, basis))
    ortho = gramschmidt(basis)

    def mu(i, j):   #returns gram-schmidt coeffs
        return ortho[j].proj_coff(basis[i])

    k = 1
    while k < n:
        for j in range(k - 1, -1, -1): #check for all j< k
            mu_kj = mu(k, j)
            if abs(mu_kj) > 0.5: # size condition
                basis[k] = basis[k] - basis[j] * round(mu_kj)
                ortho = gramschmidt(basis)

        if ortho[k].sdot() >= (delta - mu(k, k - 1)**2) * ortho[k - 1].sdot(): #lovasz condition
            k += 1
        else:
            basis[k], basis[k - 1] = basis[k - 1], basis[k] #swap both
            ortho = gramschmidt(basis)
            k = max(k - 1, 1)

    return [list(map(int, b)) for b in basis]
```

Implementation(Improved)

- In the original code, After the correction due to size condition we are updating the orthogonal matrix.
- But we have observed the orthogonal matrix does not change due to the operation

    basis[k] = basis[k] - basis[j]*round(mu_kj)

- So we have updated the code so that orthogonal matrix calculation is skipped for size condition.
- Also, For orthogonalization if Lovasz condition fail, we only need to update k-1 and k^{th} vectors in the orthogonal matrix since only $(k-1)^{th}$ and k^{th} vectors were swapped.
- This have improved runtime performance by a heavy margin.

Implementation(Improved)

```
def reduction1(basis, delta = 0.75) : #our main func
    n = len(basis)
    basis = list(map(Vector, basis))
    ortho = gramschmidt(basis)

    def mu(i, j):   #returns gram-schmidt coeffs
        return ortho[j].proj_coff(basis[i])

    k = 1
    while k < n:
        for j in range(k - 1, -1, -1): #check for all j< k
            mu_kj = mu(k, j)
            if abs(mu_kj) > 0.5: # size condition
                basis[k] = basis[k] - basis[j] * round(mu_kj)

        if ortho[k].sdot() >= (delta - mu(k, k - 1)**2) * ortho[k - 1].sdot(): #lovasz condition
            k += 1
        else:
            basis[k], basis[k - 1] = basis[k - 1], basis[k] #swap both
            ortho = gramschmidt_up(ortho,k,basis)
            k = max(k - 1, 1)

    return [list(map(int, b)) for b in basis]
```

Implementation(Improved)

```
A = np.random.randint(-500,500, size=(10,10));A
```

```
array([[ 109,  341, -322,   79,  -77,  468, -487, -124,  -57, -151],
       [-146, -133,  440,  117, -106, -224,  490,  306, -415,  394],
       [-408,   33,  481, -208,   62, -161,  -64, -468,  188,  174],
       [-441, -453, -488, -166,  409,   -3, -478,  442,   -1, -215],
       [-224,  255,  100,  -74,  464,  280,  -37,  419, -102,    9],
       [ 139,  333,  -65,  375, -258,  270,  172,   53,   49, -207],
       [ 103, -331,   99, -312,  -73, -463, -117,  -19,  276,  259],
       [ 143, -438, -303, -350, -283,  364,  167, -110, -215,  429],
       [ 459, -233, -173, -401, -208, -220, -442,  126,  358,  342],
       [ 398,  -50, -168,   21,  314, -368,  165,   -5, -455,  245]])
```

```
print(timeit.timeit('reduction_(list(A))', globals=globals(), number=1)*100)
```

```
4.0514125999834505
```

```
print(timeit.timeit('reduction(list(A))', globals=globals(), number=1)*100)
```

```
21.283085300092353
```

```
np.array_equal(np.array(reduction(list(A)))-np.array(reduction_(list(A))),np.zeros((10,10)))
```

```
True
```

- The code block shows the drastic increase in performance. (20 s -> 4 s)
- It also shows the correctness by comparing them.

Runtime

- It’s polynomial time algorithm
- Given a basis with d linearly independent n-dimensional integer coordinates vectors, algorithm calculates reduced lattice basis in time

$$O(d^5 n \log^3 B)$$

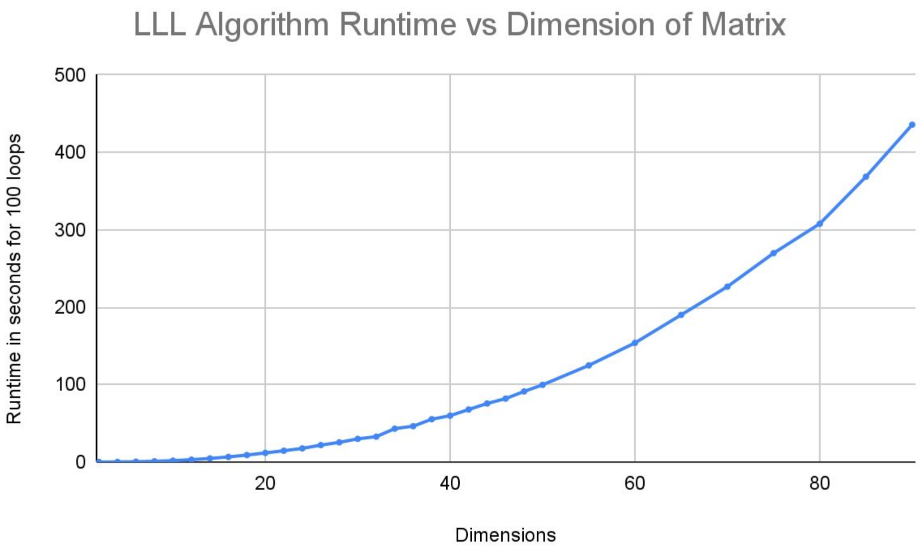
- where B is largest vector in basis under Euclidean norm.

Runtime vs Dimensions of matrix analysis

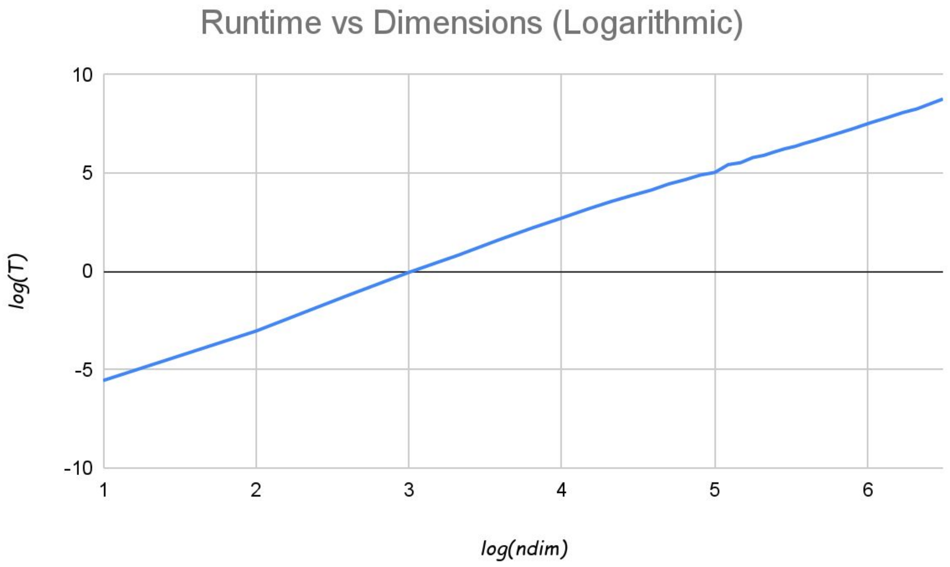
```
for size in range(2, 50, 2):
  print(size,timeit.timeit('reduction_(list(np.random.randint(-500,500, size=(size,size))))', globals=globals(), number=100))
```

- For each size of matrix, we will create an array of random integers from the range [-500, 500] uniformly.
- Then do LLL reduced basis algorithm and note the runtime after repeating the process 100 times.
- We change the size in steps of 2 and after crossing 50, we will change in steps of 5 (Because it's too time consuming).

Runtime vs dimensions



Runtime vs Dimensions



- We plotted log(T) vs log(ndim) for finding exponent of polynomial time LLL algorithm complexity.
- Slope was around 2-2.5 (Worst case is 6 theoretically)

Perturbations to inputs

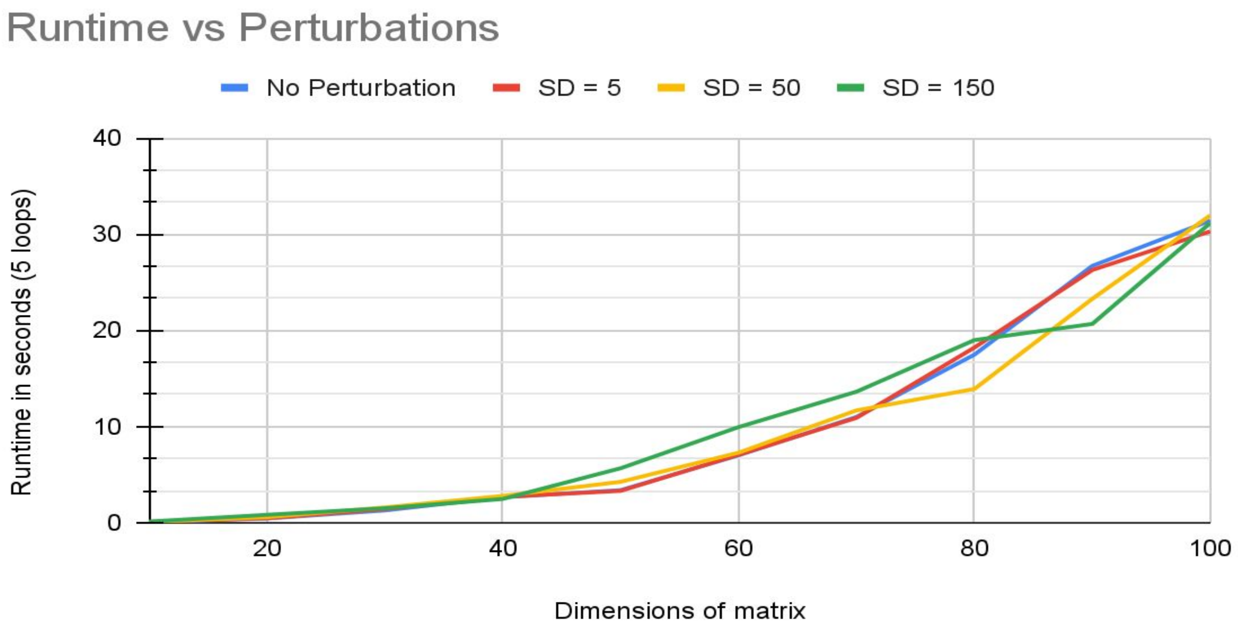
- For perturbation analysis, we will add an array of random numbers devised from gaussian distribution with zero mean and analyze for different deviations.
- We have to keep in mind that it’s an integer algorithm, so we need to round it to nearest integers after perturbing.
- Set of deviations we tried: 5, 50, 100, 150

Perturbation Code

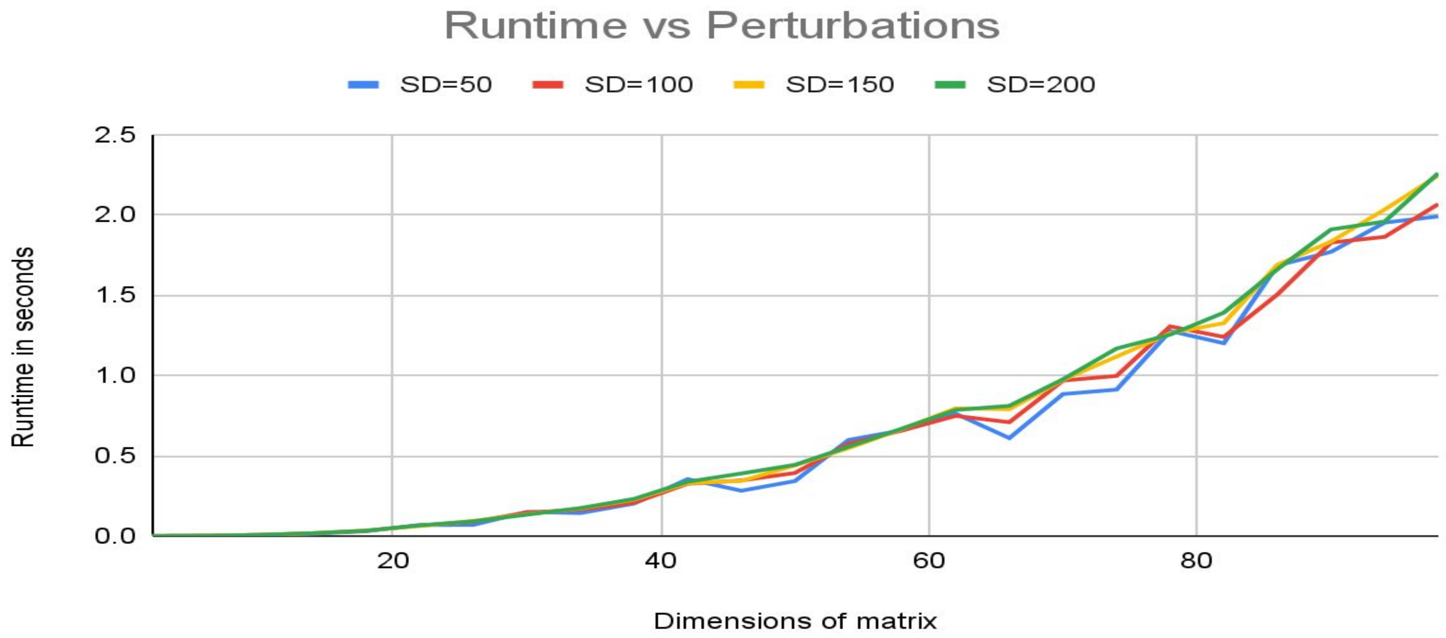
```
variances = [0.05, 0.1, 0.15]
for dim in range(10, 110, 10):
  print('n =',dim)
  A = np.random.randint(-500,500, size=(dim,dim))
  print('Without Perturbation',timeit.timeit('reduction_(list(A))', globals=globals(), number=1)*100)
  for var in variances:
    print('Perturbation with variance', var,':',timeit.timeit('reduction_(list(A+ np.round( np.random.normal(0, var*1000, size = (dim, dim))).astype(int)))',
                                                     globals=globals(), number=100))
```

```
n = 10
Without Perturbation 1.5325167000128204
Perturbation with variance 0.05 : 2.143665745000135
Perturbation with variance 0.1 : 2.0126204690000122
Perturbation with variance 0.15 : 2.0071849989999464
n = 20
Without Perturbation 22.00010049998582
Perturbation with variance 0.05 : 14.659756155000196
Perturbation with variance 0.1 : 13.490080590999924
Perturbation with variance 0.15 : 11.906798750000007
n = 30
Without Perturbation 36.89423319999605
Perturbation with variance 0.05 : 32.34531388999994
Perturbation with variance 0.1 : 33.28070798199997
Perturbation with variance 0.15 : 36.56957571199996
```

Perturbation results (5 loops)



Perturbation results (100 loop average)



Analysis

- Algorithm runtime not only depends on dimensions, but it also depends heavily on the relation between vectors (Degree of Orthogonality).
- If also depends on initialization of random numbers. If the vectors of integers generated are close to orthogonality, it takes less time, otherwise more.
- From the above graph, we can observe that runtime doesn’t change much for perturbation with standard deviation 5, but for deviation of 150, as there is a possibility of drastic degree of orthogonality change, runtime changes a lot.
- In general as standard deviation increases runtime increased(There were some exceptions)

Thank You